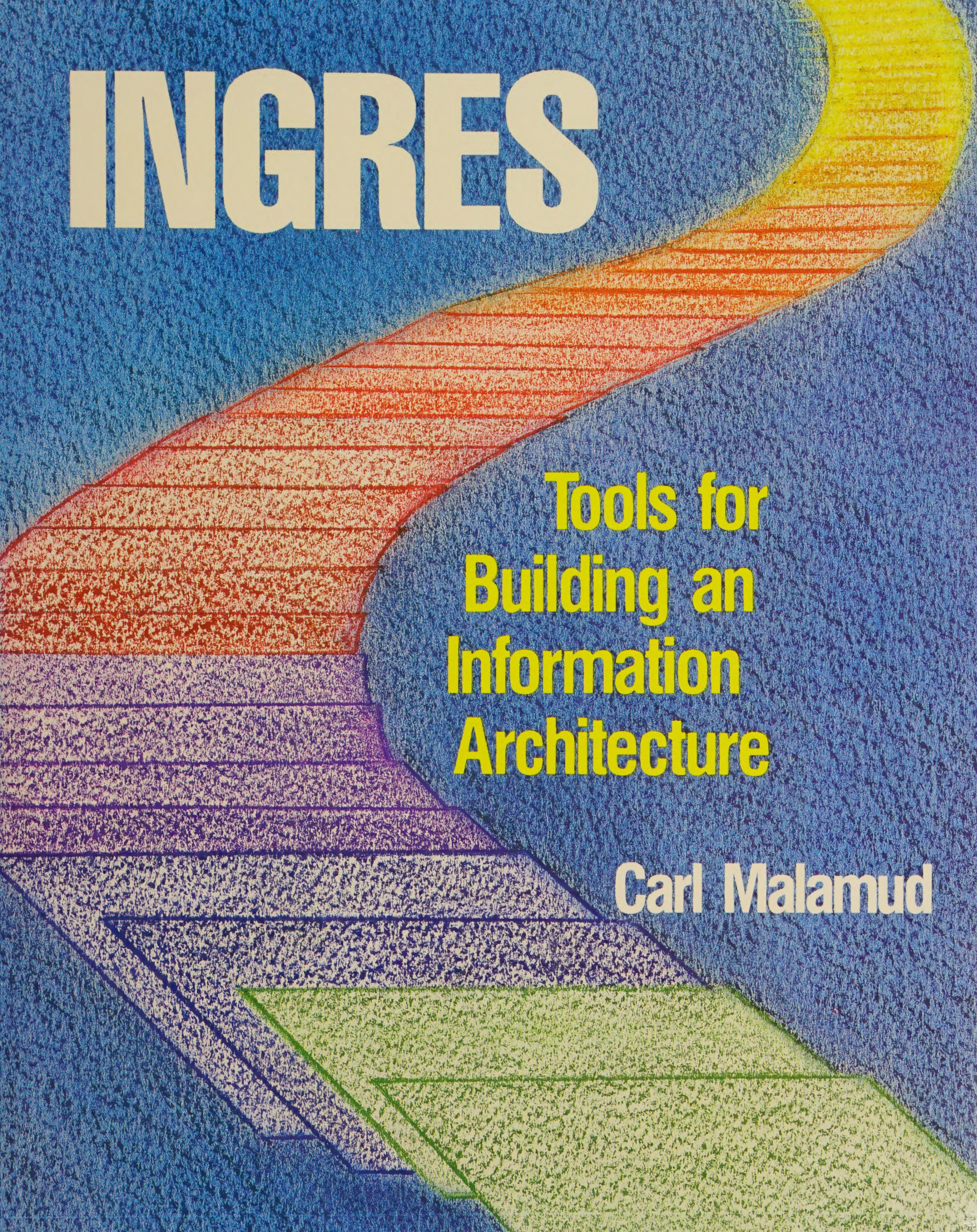
INGRES

Tools for Building an Information Architecture

Carl Malamud



INGRES

Tools for Building an Information Architecture

Carl Malamud

The INGRES relational database system is perhaps the most powerful and sophisticated of its kind. However, because so much of the related literature has focused on theory and the Structured Query Language (SQL), data processing professionals have not always been able to unleash the problem-solving power of INGRES and other information systems. This unique book is the practical guide they have long awaited.

Carl Malamud, a certified INGRES instructor, has worked with INGRES since the first version appeared commercially. He presents INGRES within the vital context of solutions to business problems – focusing on user interfaces, data repositories, and their integration with computer systems and networks.

With this concise volume, readers have at their fingertips virtually every tool needed to build a productive information environment. Malamud clearly explains how to develop front-end user interfaces...administer and fine-tune back-end data servers...establish network architectures...and tap into unique INGRES features that affect physical storage, query optimization, locking, and performance.

In the crucial area of networking, Malamud alerts readers to the latest in distributed databases and gateways to other systems. And in describing how to manage the application development process, he outlines the newest and best uses of data dictionaries and computer-aided software engineering (CASE) tools. Readers will also

(Continued on the back flap)

A VAN NOSTRAND REINHOLD BOOK

To Dr. Malamud:

Congratulations on
getting your Ph.D.
I'm proud to have
a mother with an
advanced degree
in bugs.

Love,

INGRES

Tools for Building an Information Architecture

Carl Malamud

VNR Van Nostrand Reinhold
New York

Library of Congress Catalog Card Number 89-5344
ISBN 0-442-31800-6

Printed in the United States of America

Designed by Carl Malamud

Van Nostrand Reinhold
115 Fifth Avenue
New York, New York 10003

Van Nostrand Reinhold International Company Limited
11 New Fetter Lane
London EC4P 4EE, England

Van Nostrand Reinhold
480 La Trobe Street
Melbourne, Victoria 3000, Australia

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario, M1K 5G4, Canada

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Malamud, Carl, 1959–
INGRES: tools for building an information architecture.

Includes index.
1. Data base design. 2. INGRES (Computer system)
I. Title.
QA76.9.D26M35 1988 005.75'65 89-5344

This book is dedicated with all my love to Dr. Jean G. Malamud.

Contents

PART I: The User Interface

PART II: The Data Manager

PART III: Remote Data Access

PART IV: Managing Development

Preface

This is a book about the INGRES relational database environment. It is tempting in writing a book about databases to devote many of the pages to a discussion of the Structured Query Language (SQL), the fundamental language used to retrieve and modify data in the database.

Talking about INGRES solely in the context of SQL, is like discussing personal computers in the context of the BASIC programing language. Granted, BASIC and other programming languages are important building blocks in computing. Most users, however, do not see programming languages. They see applications—word processors, graphics, or other solutions to their business problems.

This book attempts to present INGRES in the broad context of information systems and solutions to problems. Instead of SQL, the book focuses on user interfaces and data repositories. The first two parts of the book discuss these issues.

User interfaces are often general-purpose user interfaces, that is, data browsers that work on any data in the database. These general-purpose user interfaces require no programming. Query-By-Forms is an example of such an interface. Part I of the book discusses these general-purpose applications in traditional forms-oriented terminals, as well as in a windowed environment on workstations.

Part I also discusses application development. Fourth-generation languages (4GLs) are high-level tools used to develop a new application quickly. Whereas Query-By-Forms is often appropriate simply for data entry or retrieval, 4GL-based applications are capable of providing highly sophisticated information systems.

Part II of the book discusses the data manager. The user interface has the important characteristic of not knowing or caring how the underlying data are stored, or where they are stored. The data manager has the job of translating a logical request for data into efficient retrieval of large amounts of information.

Part II discusses the techniques used in the data manager to retrieve large amounts of information efficiently, first in a single-user environment, then in an environment with concurrent access to data by many users. Finally, extensions to the data manager currently under development in university research environments are discussed to provide a glimpse of what tomorrow's database management systems (DBMS) may look like.

Part III deals with information systems in a distributed, networked environment. Very few computing networks consist of a single database. Instead, data are distributed in a large variety of data repositories. Some of these are INGRES databases; others may be other commercial database management systems. This section focuses on how a single user interface is able to access these different repositories as if they were a single logical database.

Part IV discusses how information systems are developed. Data dictionaries and computer-aided software engineering (CASE) are tools for managing large, integrated information systems. These systems may be INGRES databases and applications, but probably also have components from other subsystems. Part IV ends with a discussion of how all these different components fit together to make up an information architecture—a framework used to manage changing needs for information in a complex computing environment.

A Note on Trademarks

- Applications-By-Forms (ABF), Forms Run-Time System (FRS), INGRES, INGRES Database Gateways, INGRES Data Manager, INGRES/EQUEL, INGRES/ESQL, INGRES/FORMS, INGRES/Gateways, INGRES/Gateway to Rdb, INGRES/Gateway to RMS, INGRES/MENU, INGRES Multi-Server Data Manager, INGRES/NET, INGRES Query Optimizer, INGRES/STAR, INGRES/STAR Distributed Data Manager, Query-By-Forms (QBF), Report-By-Forms (RBF), Star*View, Visual-Forms-Editor (VIFRED), Visual-Graphics-Editor (VIGRAPH) are trademarks of Relational Technology, Inc.
- CADRE, team*work*, team*work*/SA, team*work*/RT, team*work*/SD, team*work*/ACCESS and team*work*/IM are trademarks of CADRE Technologies Inc.
- INGRES/team*work* is a trademark of Relational Technology and CADRE Technologies Inc.
- Unix is a trademark of AT&T.
- 20/20 is a trademark of Access Technology.
- PostScript is a trademark of Adobe Systems, Inc.
- Apple and AppleTalk are registered trademarks of, and Macintosh is a trademark licensed to, Apple Computer, Inc.
- dbase is a trademark of Ashton-Tate.
- RS/1 is a trademark of BBN Software Products.
- IDMS/R and Cullinet are trademarks of Cullinet.

- DEC, EMA, VAX, VAX Cluster, VMS, RMS, Rdb, Ultrix are trademarks of Digital Equipment Corporation.
- IBM, IBM PC, DB2, IMS, MVS, VM/CMS, VSAM are trademarks of International Business Machines Corporation.
- ART is a trademark of Inference Corporation.
- IntelliCorp and KEE are registered trademarks, and KEEconnection is a trademark, of IntelliCorp.
- Natural Language and NLI are trademarks of Natural Language, Incorporated.
- Lotus 1-2-3 is a trademark of Lotus Corp.
- MS-DOS is a trademark of MicroSoft.
- Multiplex and CL/1 are trademarks of Network Innovations.
- WordPerfect is a trademark of WordPerfect Corporation.
- Carl Malamud is a trademark of Carl Malamud.

Acknowledgments

This book would not have been possible without the extensive cooperation of Relational Technology, both as a corporation and the individuals that make up the company. After eight years of working with INGRES as a product, I am still impressed by the sophistication of INGRES and of the depth and breadth of knowledge of the employees of RTI.

In the course of doing research on this book, many individuals in RTI and at other companies have willingly provided their time and expertise. I am particularly indebted to the following RTI employees: Paul Butterworth, Richard Desmond, Olivia Dillan, Derek Frankforth, Neil Goodman, Mark Hanner, Diane Harvey, Robert Healy, Ed Horst, David Hung, Carol Joyce, Robert Kooi, Peter Madams, Alison Martin, Robert McCord, Robert McQueer, Juan Montez, Paul Newton, Joann Rice, Peter Schmitz, David Simonson, Daniel Tyack, Jackson Warden, Jr., Glenn Winokur, Jonathan Wong, Derek Wright, and Aaron Zornes.

I would also like to express my thanks to Professors Lawrence A. Rowe and Michael Stonebraker of the University of California at Berkeley. Their careful reviews of the manuscript and their patient explanations of many key concepts are greatly appreciated. I was also aided by the following students and staff members of the university on the sections on the Picasso and Postgres projects: Sharon Wensel, Mike Hirohama, Yong Dong Wang, Brian Smith, and Luis Miguel.

Richard Berger and Eric Reid of Sun Microsystems, Eric Weiner of IntelliCorp, and John Manferdelli and David Coleman of Natural Language, Incorporated provided valuable help and resource materials for their products.

Finally, I'd like to thank Dianne Littwin, my editor at Van Nostrand Reinhold.

Chapter

1

Tools for Information Systems Productivity

This is a book about information architectures—a way of assuring that people can find information on a computer network easily and efficiently. The computer network can be made up of many different brands of computers and database systems. This book shows how different computers and database systems can be organized to provide convenient access to information.

The reason for an information architecture is user productivity. End users need a variety of decision support tools that allow them to find information and display it in a usable format. The information architecture ensures that different tools, such as report writers and spreadsheets, are all able to access data without having to know where or how they are stored.

An information architecture is also used by programmers, who need tools that allow them to quickly develop applications that solve specific problems for an organization. For example, an accounts receivable application manages data about the money owed to the organization. An application such as this usually provides a custom interface for each class of users, such as accounting clerks or supervisors.

In order for programmers to be productive, they need to be able to develop applications that are not tied to a particular type of computer or database management system (DBMS). Computers change quickly, and if the application only runs on one type of computer, it will have to be rewritten. A portable application allows the organization to preserve the investment in software even though the computer or DBMS is changed. Portability of tools is a key component of an information architecture.

Data in any organization reside in a variety of different places, known as data repositories. These data repositories could be a file or a DBMS. All of the tools, whether for application development or end-user decision support, must be able to access these heterogeneous data repositories without worrying about the location or the type. To provide access to heterogeneous data repositories, the information architecture needs to provide two key features. First, information has to be accessible over a complex computing environment consisting of different kinds of computers and different kinds of networks. The user should not have to navigate the network or the computer system—a tool should work the same no matter what data it accesses.

The second key feature is that users should be unaware of the kind of data repository they are accessing. The tool should work the same on a file as on an INGRES database. To the user, these different data repositories are integrated into one transparent database. The result is that users ask for data. The location of the data, the brand of computer, the brand of database system, or the particular database that the data resides in should all be transparent.

An information architecture thus consists of a series of integrated tools able to access the same information. The tools vary in power and complexity depending on the type of user, but they all work together. Information can be moved from a database into a report and then into a word processor without worrying about the mechanics of the transfer. The user is thus freed to concentrate on the task at hand. For an end user, the task is making decisions based on information. For the programmer, the task is to develop applications. Integration of data in a transparent fashion over a heterogeneous, distributed computing network is the goal.

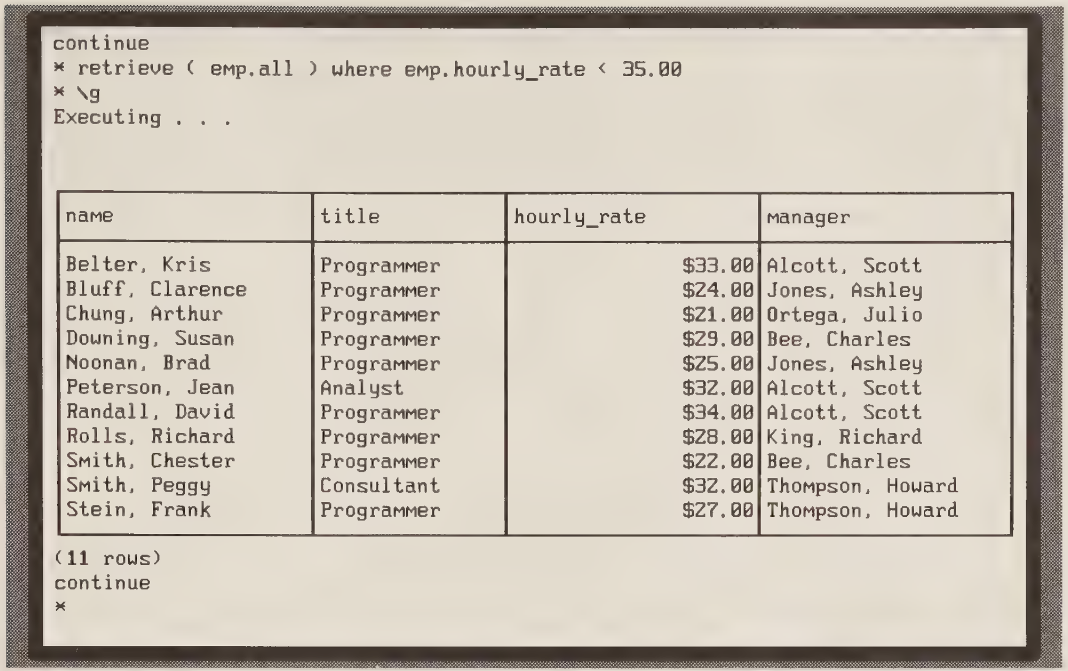
This book looks at the key components of that information architecture. This chapter provides a survey of the major components, and the remainder of the book provides an in-depth examination of each component.

Front- and Back-End Processing

Most modern DBMSs are divided into two components: the front end and the back end. Each of these components is a separate program or collection of programs. The two components use a query language to communicate requests for data. The front end is responsible for managing the user interface. This involves presenting information on the screen and providing help services and other components usually seen by the user.

There are various types of front ends available in the INGRES environment, each presenting a different type of user interface. These front ends are called applications. One application, VIGRAPH, presents a graphical version of information. Another, the report generator, provides a more traditional report of information. Query-By-Forms (QBF), a third application, is used for interactive browsing and modification of data. All three of these applications are general-purpose user interfaces. Once users know how to use them, they are able to use the tool on any table in the databases they access.

Many computing environments have custom front ends. These front ends are developed for a specific task, for example, an order entry system. INGRES provides a set of tools for developing these custom interfaces.



```
continue
* retrieve ( emp.all ) where emp.hourly_rate < 35.00
* \g
Executing . . .


name                 title            hourly_rate           manager

Belter, Kris         Programmer                     $33.00  Alcott, Scott
Bluff, Clarence      Programmer                     $24.00  Jones, Ashley
Chung, Arthur        Programmer                     $21.00  Ortega, Julio
Downing, Susan       Programmer                     $29.00  Bee, Charles
Noonan, Brad         Programmer                     $25.00  Jones, Ashley
Peterson, Jean       Analyst                        $32.00  Alcott, Scott
Randall, David       Programmer                     $34.00  Alcott, Scott
Rolls, Richard       Programmer                     $28.00  King, Richard
Smith, Chester       Programmer                     $22.00  Bee, Charles
Smith, Peggy         Consultant                     $32.00  Thompson, Howard
Stein, Frank         Programmer                     $27.00  Thompson, Howard

(11 rows)
continue
*
```

Courtesy of Relational Technology

Fig. 1-1 Retrieving Information Using QUEL

The back end is the data manager. In the INGRES environment a single back-end server is able to accept requests from multiple users (technically from multiple front ends). This back end is known as a data server. It is the responsibility of the data server to efficiently retrieve data for users while preserving the integrity of the database. The first two parts of this book will discuss the front and back ends in turn.

A query language is used for communication between front and back ends. INGRES uses two different query languages:

- query language (QUEL)
- structured query language (SQL)

QUEL is the original language that was supported in an INGRES database. In fact, in the original versions of INGRES, the only way to retrieve data was to formulate QUEL commands. For example, to retrieve all the information in a table called emp, the user would submit the following query:

```
retrieve ( emp.all )
```

Figure 1-1 shows the result of this query. Notice that the query retrieves a table of data. Tables are the fundamental construct in a relational database system such as INGRES. The *emp* table has a series of columns, each one containing a different piece of information. The table also has a series of rows, one for each instance of an employee.

The relational model provides the user with access to data in a logical fashion. Users ask for certain rows and columns of a table. This is in contrast to more traditional file management systems or older database systems where a programmer has to write a small program. In the case of the file, the programmer would have to open the file, declare variables, write routines to read the data, and then write routines to display the data. All of these functions are transparent to the user in a relational database and are handled by the back end.

More general user interfaces, such as QBF, allow the user to ask for data in a more intuitive fashion. When the user types "GO" (retrieve the data), the user interface formulates a QUEL (or SQL) command and sends it off to the back-end data manager. The data manager then returns either an error message or a set of data. Figures 1-2 and 1-3 show the QBF equivalent of the QUEL command and the resulting data.

Both SQL and QUEL were the result of research projects conducted in the early 1970s. QUEL was the query language used in the University INGRES project at the University of California at Berkeley. SQL was the result of the System R project conducted at IBM. Both projects were an attempt to solve major problems in the early relational DBMSs. First, they both developed non-procedural query languages for access to data in a logical fashion. Both projects also worked to develop algorithms to optimize queries so that data could be retrieved quickly.

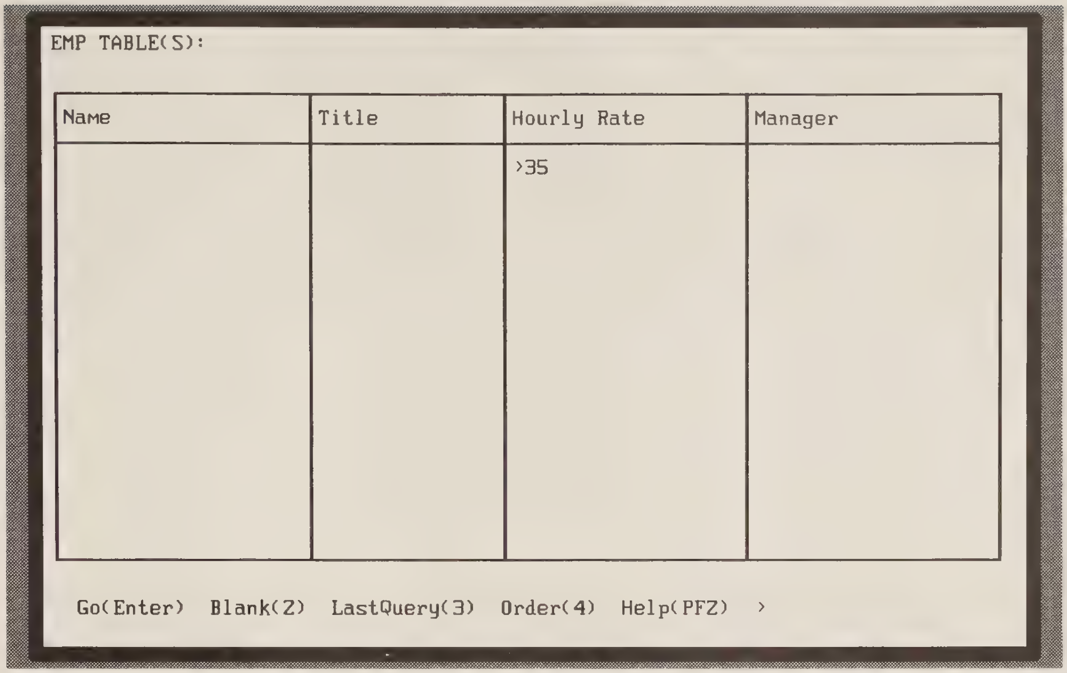
Both SQL and QUEL have advantages and disadvantages. Many people believe that QUEL is a superior query language because it is more consistent and more powerful than SQL. On the other hand, many people think that simple queries are easier to express in SQL. Technical merit notwithstanding, SQL was adopted by IBM as their standard query language. Following that, SQL has been adopted as both a national and international standard. Relational Technology fully supports both query languages with INGRES databases.

Because of the standard status of SQL, it will be used for most examples in this book. However, in the chapter on Postgres, a university research project, QUEL will be used for examples since it forms the basis for that data manager.

With SQL, as with QUEL, end users rarely use the query language directly. A query language requires the user to carefully specify the desired operation with the correct syntax. Instead, a more intuitive interface, based on a visual representation of the data, will be used, as in the case of a form that allows a user to see the fields in a table and fill in the values on the fields he is interested in.

The SQL language consists of different classes of commands. Data manipulation commands are used to access and change data in the database. Data definition commands are used to create and change tables in the database as well as to create storage structures and indices for the data that alter the performance of various types of queries. Data manipulation commands allow users to retrieve, update, delete, or add new data. These are the basic SQL statements that will be seen in most of this book. An example of a command to retrieve data from the database is the following *select* command:

```
select * from emp where emp.hourly_rate < 35.00
```



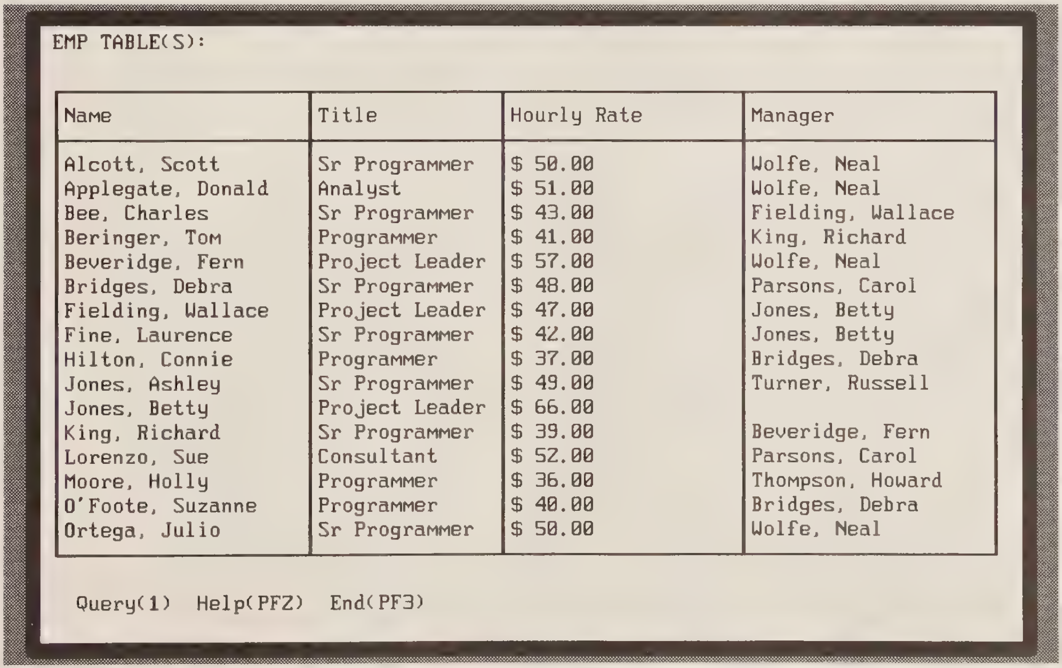
EMP TABLE(S):

Name	Title	Hourly Rate	Manager
		>35	

Go(Enter) Blank(2) LastQuery(3) Order(4) Help(PF2) >

Courtesy of Relational Technology

Fig. 1-2 QBF Equivalent of a QUEL Query



EMP TABLE(S):

Name	Title	Hourly Rate	Manager
Alcott, Scott	Sr Programmer	$ 50.00	Wolfe, Neal
Applegate, Donald	Analyst	$ 51.00	Wolfe, Neal
Bee, Charles	Sr Programmer	$ 43.00	Fielding, Wallace
Beringer, Tom	Programmer	$ 41.00	King, Richard
Beveridge, Fern	Project Leader	$ 57.00	Wolfe, Neal
Bridges, Debra	Sr Programmer	$ 48.00	Parsons, Carol
Fielding, Wallace	Project Leader	$ 47.00	Jones, Betty
Fine, Laurence	Sr Programmer	$ 42.00	Jones, Betty
Hilton, Connie	Programmer	$ 37.00	Bridges, Debra
Jones, Ashley	Sr Programmer	$ 49.00	Turner, Russell
Jones, Betty	Project Leader	$ 66.00	
King, Richard	Sr Programmer	$ 39.00	Beveridge, Fern
Lorenzo, Sue	Consultant	$ 52.00	Parsons, Carol
Moore, Holly	Programmer	$ 36.00	Thompson, Howard
O'Foote, Suzanne	Programmer	$ 40.00	Bridges, Debra
Ortega, Julio	Sr Programmer	$ 50.00	Wolfe, Neal

Query(1) Help(PF2) End(PF3)

Courtesy of Relational Technology

Fig. 1-3 The Results of the QBF Search

This command gets all of the columns from a table called emp and selects all rows of that table having an hourly rate greater than 35. More sophisticated versions of the select command allow users to look only at certain columns of data, or to group and aggregate data. For example, the user could request the names of all employees in departments with an average salary greater than $10,000.

The other class of commands is the data definition commands. A simple example is the *create table* command, which creates a new table in the database. The user defines certain columns for this table and assigns a data type to each of the columns. For example, to create a table called emp with four columns, the user would submit the following:

```
create emp ( name = char(16), number_years = integer ,
                     salary = money, date_hired = date )
```

Each of the columns in the emp table is assigned a data type. The name column is a sixteen-character text string. The number of years column is an integer. The table definition also includes two other data types: date and money. These are known as abstract data types because they are not native to the operating system that INGRES runs on.

Internally, date is stored as an integer, representing the number of seconds since an arbitrary starting point. To users, though, dates appear in a format that they are familiar with. INGRES converts the internal representation of date into an external representation. Users can then perform various operations on columns of type date. For example, a user can request all columns where the date the employee was hired is less than "today" minus "1 year." Notice that INGRES is able to perform date arithmetic ("today" minus "1 year") and is able to compare dates, just like it can perform integer arithmetic and comparisons.

A special class of data type in INGRES is data of type procedure. A procedure is a collection of SQL statements that are automatically executed when the procedure is accessed. A procedure might be used to change several different tables in the database. For example, whenever a new sale is registered in the database, the user could execute a procedure called *Register_Sale*. Register_Sale recalculates summary sales figures, such as cumulative sales to date and puts that information in a summary table.

Procedures in INGRES are not part of normal tables, but are all kept in a special table. A user cannot define a column Register_Sale as part of the emp table. Instead, the information is kept in the special procedures table. We will see in Chapter 7 in the discussion of the Postgres that one of the extensions to the relational model provided by this research project is to allow users to define data in normal tables of type procedure.

This book does not discuss the SQL or QUEL languages extensively, it is instead concerned with broader issues—the design of data servers and user interfaces and the development of an information architecture that allows the components to communicate. This is not to say that SQL or QUEL are not important topics, only that they are left to other books in the field. Readers interested in a more in-depth discussion of SQL are directed to C. J. Date, *An Introduction to Database Systems,* Volume I (4th ed., Addison Wesley, 1986, Reading, Mass.).

Some users never need to learn a query language. End users will use a series of decision support tools such as QBF to perform most of their work. It is fairly rare that a sales manager, for example, would want to use SQL, FORTRAN, or any other language. Programmers, on the other hand, will often embed SQL into programs. In this case, the database is being used in place of a traditional file system. Instead of opening files and reading records, the programmer requests data in the language of SQL. Data retrieved are then put into programming language variables and the program continues to execute.

The reason a programmer uses a database instead of a more traditional file is that the database is easier to use. Instead of concentrating on writing code to open and close files, the programmers can concentrate on developing an application for their clients. Another advantage of the database is that it takes care of ensuring that data stay consistent when multiple users are trying to access the same information.

End users will thus often use an intuitive interface such as QBF. Programmers will also use tools like QBF on occasion. Programmers, however, will supplement those tools with a set of languages such as SQL and other programming languages for developing complex applications. As will be seen in Part II of this book, there are a variety of different classes of tools available spanning different levels of complexity and functionality.

Tools for the User Interface

Part I of this book discusses how the user interface is constructed. The first chapter, Chapter 2, discusses general-purpose user interfaces. A general-purpose interface is an application that can be used to examine data for a variety of different users. An example used on personal computers is the Lotus 1-2-3, which is a general purpose spreadsheet package. Once users know how to use Lotus, they are able to apply it to a wide variety of different applications.

In INGRES, there are three general-purpose user interfaces available:

- Query-By-Forms (QBF)
- Report-By-Forms (RBF)
- Visual-Graphics-Editor (VIGRAPH)

All three interfaces use a form as the means of interacting with the user. The form is a display on the screen with regions for data display or input, as shown in Figures 1-2 and 1-3. The form also has a menu associated with it. Users pick menu items to perform certain tasks. For example, the *HELP* menu item is used to find out more information on what to do.

Query-By-Forms (QBF) provides an interactive method of browsing, appending, or updating data in the database. When users start QBF, they are able to view a catalog of data available, known as query targets. After picking a query target, users can perform various queries to examine and manipulate the data.

Report-By-Forms (RBF) is the second general-purpose user interface in INGRES. RBF lets the user quickly generate a format for a report. A template of the report is put on the screen, and the user can change various components, such as the column titles. Users can also perform more sophisticated operations such as having aggregates automatically calculated. For example, in a sales report, the user could specify that the sum of sales for each department appear at the bottom of each page.

The third interface is the Visual-Graphics-Editor (VIGRAPH), which allows users to graphically display data. Instead of a report, the user can examine a bar graph or pie chart. VIGRAPH allows the user to edit the graph. The definition of the graph is stored in the database, and whenever the graph is run, the latest version of the data are retrieved from the database and the graph is displayed.

Associated with the three user interfaces are a wide variety of different utilities. The Visual Forms Editor (VIFRED) is used to develop custom forms. VIFRED can be used to add explanatory text to the form or to change the visual characteristics of fields. VIFRED can also be used to add more sophisticated capabilities such as a validation check that ensures that the proper data are entered in a particular field.

Other utilities include a variety of catalogs. After a report has been saved in RBF, users can consult the catalog of available reports. The available reports are displayed on a form. The user moves the cursor to the report and picks a menu item to run the report.

Another forms-based utility is Application-By-Forms (ABF). ABF is used to develop custom interfaces, which are applications designed to solve a particular problem. While QBF could be used for a data entry system, the facilities of ABF provide a more focused user interface for the application. Chapter 3 discusses the application development facilities available in INGRES.

Within ABF, the application developer has access to all the facilities in the general-purpose user interfaces. The developer might use ABF to construct a main menu for the application. One of the options, such as *Add_Data,* could then call QBF, using a particular form, in append mode.

In addition to using the services of the general-purpose subsystems, the developer can write new functions using the INGRES Fourth Generation Language (INGRES 4GL). INGRES 4GL is a high-level language, meaning that a few lines of code accomplish a great deal. INGRES 4GL is used to control the interaction of the user with the application, to manipulate data in the database, and to manipulate the form on the user's display. ABF handles many of the details of programming, leaving the developer free to concentrate on the task at hand instead of performing housekeeping tasks.

It is also possible to embed SQL and forms-manipulation commands into a traditional programming language. If the programmer was developing a statistical analysis system in FORTRAN, for example, the database could be used to store data. The programmer can use SQL statements to retrieve data instead of having to worry about manipulating files and records. The programmer can also use forms to display data on the screen instead of writing a display program from scratch.

Chapter 4 discusses several efforts currently under way to extend the user interface. Simplify is a joint effort between Sun Microsystems and Relational Technology to bring general-purpose user interfaces to workstations. Workstations have more sophisticated graphics capabilities than more traditional character-oriented terminals. Simplify exploits the capabilities of the workstation to provide a more intuitive and powerful user interface.

The last part of Chapter 4 discusses the Picasso project at the University of California at Berkeley. Picasso is aimed at the application developer and provides programmers with a more sophisticated environment. For example, Picasso allows a more complex type of form consisting of mixed graphics, text, and other forms of information to be used in an application.

Data Servers: Performance

Part II discusses data servers and other back end mechanisms. Applications issue SQL or QUEL statements and deliver them to a back-end. Part II examines what happens after the queries are received.

The data server accepts requests for data in a query language. An important attribute of a query language is that users ask for data in a logical fashion. Users ask for all rows of data meeting certain criteria. The responsibility of the data server is to accept these logical requests for data and at the same time manage the environment to ensure the integrity of the database and service requests as quickly as possible. As far as users are concerned, quick access to data is often the most important criterion. A user might ask for all personnel records for employees making more than a certain amount of money. If the personnel table is extremely large, scanning the entire table could take a long time.

The first function of the data server is to store data so that they can be accessed efficiently. Direct access to data is accomplished by a series of indexes defined by the database administrator. Instead of scanning the entire table, the data server can consult the index and then directly access the relevant rows in the table.

In a complex query involving many different tables, the data server must decide not only which indices to use, but what order to process the data. The query optimizer examines a query and decides the optimal manner in which to retrieve the information. A good query optimizer examines many different potential access plans and decides which one has the best chance of getting the information quickly.

The INGRES query optimizer is an extremely sophisticated mechanism. It has the ability to take complex queries and examine a large number of potential access plans. For example, a query could specify a retrieval of all information in the manager table with managers making less than $1000 that matches rows in the employee table that have a salary greater than $2000. In other words, the user wants to see managers that make less than their employees.

The query optimizer has at least two choices to process this request. One possibility is to first go to the employee table and find employees with a salary greater than $2000. This information is stored in a temporary table. Then, the temporary table can be sorted and each row compared to the manager table to look for a match. The other possibility is to start with the manager table and find all the relevant rows and then go to the employee table looking for a match. Each of these two possibilities is known as an access plan for the data. The query optimizer has to guess how many I/O operations and how much CPU time will be taken by each of the two access plans.

The important characteristic of a query optimizer is that it cannot know ahead of time which plan is optimal. Instead, the optimizer has to guess. Say, for example, that the manager table has 1000 records and the employee table has 10,000 records. The optimizer has to look at the selection criteria, such as managers with a salary less than $1000, and guess how many rows will satisfy that condition. Depending on the selectivity of a condition, it may be more efficient to go to one table instead of the other first.

Another example of the problem with estimating the selectivity of a certain request is a query that asks for all employees having an age greater than 95. The optimizer has to choose between scanning all rows in the employee table or using an index. If this query will retrieve half the rows in the personnel table, it makes sense to bypass the index and go directly to the underlying data. On the other hand, if only a few rows will be retrieved, it makes sense to use the index, find out which rows qualify, and then access the base data.

Asking for all rows with age greater than 95 is an example of where the INGRES query optimizer is able to retrieve data efficiently. Query optimizers do not know that people don't usually work past the age of 65. This ınformation is an example of semantic information about the data. Generally speaking, query optimizers only have access to syntactic information about the data such as the data type of a column. If age is a 2-byte integer, most query optimizers will assume that ages therefore lie between 0 and 255, and therefore half the data in the base table will be retrieved.

INGRES is able to keep a statistical profile of the database that is used by the query optimizer. This profile keeps track of minimum and maximum values of data, as well as the distribution of data. If the maximum age is listed as 80 years old, INGRES knows that consulting the index will be a better strategy than scanning the entire table. If the maximum age is 140 (say our company makes yogurt), INGRES can look at the distribution of ages. The distribution table will show that a significant percentage of ages are greater than 95 and that a scan of the entire table would be appropriate.

In addition to efficiently retrieving data for single users, the data server is responsible for maintaining data integrity in a multiuser environment. It is important that two people don't simultaneously try to write over the same piece of data. Data servers control access to data by restricting the types of accesses that can be made at the same time. Access is controlled by locking the data when operations such as updating information are being performed.

To illustrate the need for locking data while they are being changed, think of a banking example. If an account has a balance of $100 and two users try to withdraw $100 simultaneously from the same account, the bank will not be in business very long. If one user is changing a piece of data, such as a bank balance, the database needs to lock the data to prevent anyone else from accessing it. When a piece of data, such as a bank account, is locked, the second user has to wait. To the user, this process is transparent. If the data are locked, the second lock request is queued. As soon as the first request is finished, that lock will be released and the lock granted.

Locking can be carried out at different levels of granularity. For example, say we wish to update all employees with a last name starting with M. The data server can take out a lock on every row that satisfies that criterion. The problem with this is that many locks are hard to manage and slow down performance. The other alternative is to lock the entire table, which allows the operation to be carried out quickly, but blocks all other users from this table. Chapter 6 will discuss how efficient concurrent access to data is managed.

Finally, the data server is responsible for ensuring the integrity of data in the case of system crashes and other types of failure. In INGRES, a recovery manager constantly monitors the database to see if a particular operation on the database had to abort. After performing part of a transaction, a user could decide not to process this particular set of modifications to the databases and abort the transaction. This could also occur when the computer that the data server is running on crashed because of a power failure.

An abort has two possible implications. First, the transaction itself could have been a multistatement transaction. For example, transferring data from a checking account table to a savings account table is a transaction that should only take effect if both parts of the transaction occur successfully. If money is taken out of the checking table and not added to the savings table, our bank customers will be extremely unhappy.

The second implication of an aborted transaction is that users who have aborted may have outstanding locks on data. Other users could be waiting in line to access information. If the application associated with an aborted transaction is gone, other users might wait indefinitely.

The recovery manager constantly monitors transactions in progress to look for indications of uncompleted operations. If it finds such an occurrence, such as an aborted transaction, the recovery manager automatically reverses the effects of the aborted transaction, returning the data to its previous state, and releases any outstanding locks. The online recovery process is an important characteristic of INGRES—it is not necessary to stop all processing just because a user aborted.

Related to the recovery manager is the archiving process. The recovery manager catches errors caused by system crashes or errors. However, it is possible that the disk drive that the database is stored on is destroyed, say a disgruntled employee threw it into a lake. The archiver moves information on completed transactions into a journal file, typically stored on a different disk drive than the database tables.

The archiver takes all completed transactions and writes them into a journal file. Users also periodically perform a checkpoint of the database—a snapshot of information at a point in time. With journal files, the database administrator is able to roll the database forward from the last checkpoint until just before the bad transaction.

The first two chapters of Part II consider these single-user and multiuser issues in turn. The last chapter of Part III, Chapter 7, is a discussion of what data servers might look like in five years. This chapter is a discussion of Postgres, a University of California research project that is exploring extensions to traditional DBMS systems to solve problems that cannot currently be solved adequately with a relational DBMS.

Postgres is discussed for two reasons. First, it is an interesting research project and provides a glimpse of the research that is taking place in the arena of database management. Second, the two professors that are conducting the research are also founders of Relational Technology. Many of their past research efforts have turned up in previous versions of INGRES. This does not mean that Postgres is Version 7 (or 8 or 9) of INGRES—only that it gives us a glimpse into the thinking of two important players in Relational Technology.

Postgres is an active database. The user can define triggers that are activated when data are changed. For example, a personnel clerk can be notified when any hourly employee has worked more than 50 hours in a week. The triggered action can update other tables in the database, notify an application that something has been changed in the database, or run an arbitrary program.

Postgres is also an extensible data server. Users can define new types of data—say box or circle—that are stored in the database and can then define new operators on these data types, such as performing a query that retrieves all boxes that overlap large circles. Most relational systems are limited in the size of data objects they can store. In Postgres, data of arbitrary size can be stored, for example, a large text or graphics file can be a piece of data in a table.

Basic data types, such as box or integer, can then be combined into complex objects—a piece of data that in turn is made up of several pieces of data. A complex object allows a simple query to be used on an abstract entity, as in the case of a column representing a submarine that is in turn made up of a large number of simple objects, such as boxes or circles. Queries can reference submarines instead of each of the individual components that make up a submarine. Objects can be structured in a hierarchy, each higher level representing a more complex concept.

One way to define a complex object is as a set of query language commands. The commands are executed whenever a piece of data in that column is referenced in a query. Because the query might return several rows or columns of data, the column is a complex object. For example, a column can be added to an employee table called "performance." When a user retrieves the performance column, the queries in that column are executed and appropriate performance data for each row in the table are returned. A salesperson may have performance based on sales versus quota, while a personnel officer may have performance based on the percentage of open jobs filled.

Postgres has many other features that are discussed in Chapter 7. In addition to extending the query language and data types, Postgres is designed to run efficiently on tightly coupled, shared memory multiprocessors. These computers, with large amounts of main memory and many CPUs, allow the query optimizer to break up a single retrieval into multiple execution plans, each running on a different processor of a parallel processor.

Remote Data Access

Part III of this book moves INGRES out of the single computer environment into a more realistic scenario—a distributed network of computers. In this environment, there are several data servers, each managing different types of data, and many different user interfaces, each running on a different computer. In a heterogeneous environment, it is important that users not be forced to navigate the network or database system to access their data. A distributed database makes all of these data repositories appear as a single logical database system. The location or type of database is transparent to the user. Rather then spend time looking for data, users can spend time making decisions.

Chapter 8 discusses a homogeneous distributed data environment—all of the subsystems are INGRES user interfaces or data servers. It begins with a discussion of the General Communication Facility (GCF), which is the method used in INGRES to shield a user interface from knowing about the location of a data manager (and vice versa). With GCF, a user interface is able to access any data repository in the network, which can consist of multiple network protocols—a heterogeneous network. A PC on a DECnet, for example, is able to access an INGRES data repository running on an IBM mainframe using a different set of networking protocols—IBM's System Network Architecture.

A single front end accessing a single data repository is an example of networked access to data (see Fig. 1-4). Users of the INGRES/NET are able to access back ends running anywhere in the network. Separating the front end from the back end is very important in a distributed network, since the user interface and the data server have very different operating characteristics. Separating them onto different computers allows each computer to be optimized for its particular task.

The next level of access is a distributed database. INGRES/STAR allows a single-user interface to access multiple data servers as if they were a single database (see Fig. 1-5). A personnel database can be on one machine, a sales database on another. Management is able to treat these multiple data repositories as a single database and can access data without knowing its location.

Chapter 9 discusses a heterogeneous data environment. A Gateway is an INGRES program used to access non-INGRES data servers. All front-end processes are able to access these heterogeneous data servers. INGRES/STAR allows INGRES and non-INGRES data repositories to be combined into a single distributed database.

INGRES/Gateways are able to access two types of heterogeneous environments:

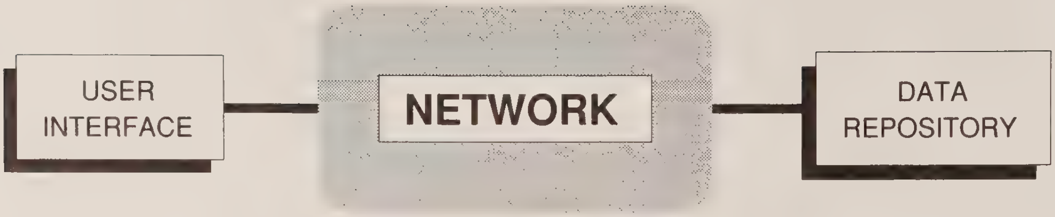


Fig. 1-4   Networked Access to Data

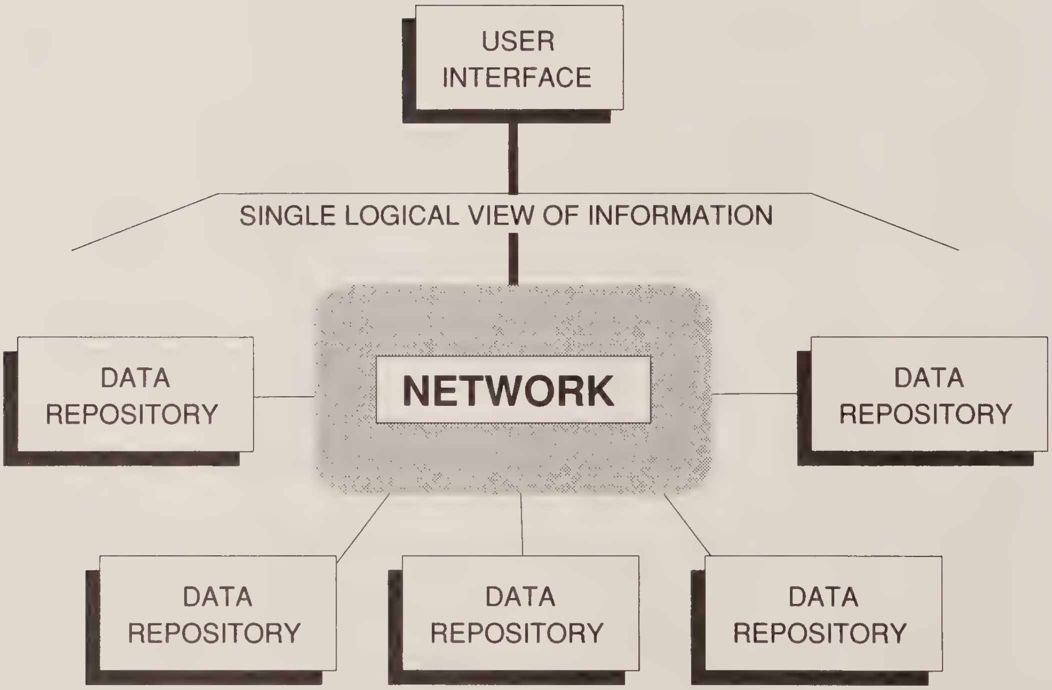


Fig. 1-5   Distributed Acess to Data

- SQL databases
- non-SQL databases

SQL databases, such as DEC's Rdb or IBM's DB2, have a large installed base. Often, there are existing applications that run in these environments. With SQL gateways, users of INGRES tools are able to use the INGRES 4GL, but still access these other environments. It is possible to access a single Rdb database, or to combine Rdb and INGRES data servers into a large distributed database. Although a large organization cannot standardize on a single DBMS to solve all problems, a gateway provides a consistent access mechanism to connect these heterogeneous systems together.

Non-SQL gateways allow more traditional data repositories, such as IBM's IMS, to be treated as a collection of database tables. Non-SQL gateways are also able to access traditional files, such as IBM VSAM files or DEC RMS files. The user can access these files using SQL statements instead of writing programs. The SQL statements are translated by the non-SQL gateway into low-level access to the data.

Using a gateway allows users to take advantage of the sophisticated user interface tools and the development environment in INGRES, but still access existing data stores. In fact, in the case of the Rdb gateway, DEC resells the INGRES tools as a supplement to their own application development tools. Gateways are also used to provide a migration path from one type of data repository into another, while still retaining access to both locations of information.

A second type of heterogeneous environment uses other user interfaces to access an INGRES data server. An example is a user doing statistical analysis. Packages like SAS are very good at statistical analysis, but are not very good at storing data. With heterogeneous front ends, users are able to store data in INGRES, but still access the data from a statistical analysis system.

A related type of heterogeneous front end allows PC- or Macintosh-based users to take advantage of the sophisticated data storage capabilities of INGRES. Users are able to retain a familiar interface, such as Lotus 1-2-3. Instead of storing the data as a Lotus worksheet, however, it is stored on a VAX on the network. Data are retrieved from the database and loaded into the spreadsheet. This type of gateway capability allows PC users to keep the tools they are familiar with, but still allows the database administrator to take advantage of the capabilities of INGRES.

Tools for Managing the Development Process

The last part of this book discusses the management and administration of a large environment that consists of many data repositories and many different types of application development efforts.

Chapter 10 discusses data dictionaries, which define what type of data are in a particular data repository. INGRES uses a data dictionary to manage itself; when a new table is added to the database, an entry is made in a data dictionary table. Then, when a user asks for data from that table, the dictionary table is consulted to find out the location of that table, security information, and a variety of other data about the data. These data about data are known as meta-data. It is the responsibility of a data dictionary to manage meta-data.

The INGRES data dictionary is stored in tables called system catalogs. These catalogs include a variety of information in addition to definitions of tables. For example, reports and forms are also stored in the system catalogs. When a user asks for a report to be run, the data server goes to the system catalogs, retrieves the definition of the report, and runs the report.

Keeping objects such as tables or reports in system catalogs has several important implications. First, the services of the data server can be used to maintain consistency. If a user is changing a report, that information is locked. Another user will then wait until the report is updated, the lock will be released, and then the new definition of the report is retrieved. The data server is also able to access these objects efficiently using the services of the query optimizer.

Another implication of storing objects as tables in the database is portability. Since reports and table definitions are just tables in the database, it is easy to move these objects to an INGRES database on another operating system. When a user moves to a new computer system, he has a choice of moving the application or database to the new computer or using INGRES/NET to access the information across the network, or some combination of the two strategies. For example, it is not unusual to move applications to a variety of different computers and have the database reside in a central computer.

The INGRES system catalogs are an efficient method of storing INGRES-related information. A more general form of data dictionary, the Information Resources Dictionary System (IRDS), is able to store any type of data and provides a series of mechanisms to control access to the data. IRDS is a data dictionary developed by standards organizations, meaning that data dictionary information from one implementation of IRDS, say in an INGRES database, can be easily moved to another implementation, say in a DB2 database (or vice versa).

IRDS is an extensible data dictionary standard. New types of information can be defined to the data dictionary, and instances of these new types of data can be stored. For example, if we supplement INGRES with a statistical analysis system, we can extend the data dictionary to include concepts such as matrix or multiregression model. We can then define specific instances of these new types of entities.

Computer-aided software engineering (CASE) is a general term for tools and methodologies used to construct very large information systems. CASE tools include application design methodologies, such as data flow diagrams, that are used to model an information system. CASE tools are then used to efficiently implement these models into working information systems.

CASE tools in the INGRES environment are based on INGRES/team*work*, a set of CASE tools developed by RTI and CADRE. These tools allow a variety of different structured development methodologies to be used to construct INGRES (and non-INGRES) information systems. INGRES/team*work* is then tied to the INGRES applications development environment to turn the logical model of an information system into a working application.

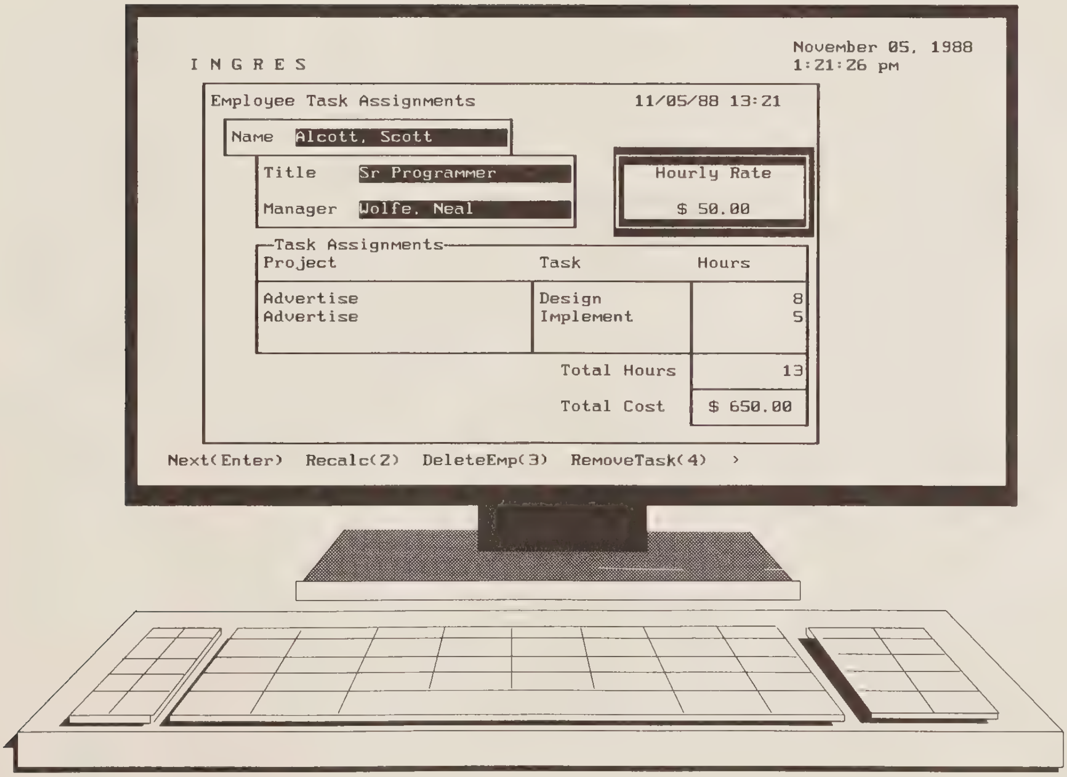
Chapter 11 discusses these various methodologies and how they fit into the INGRES/team*work* CASE tools. It also discusses the relationship of CASE tools to the rapid prototyping environment of INGRES.

Chapter 12 discusses how all of these issues can be brought together to form an information architecture for an organization. An information architecture is an attempt to deal with the distributed and dynamic aspects of today's computing environments.

It is impossible to decide exactly what an information system will look like and the underlying hardware to support it in today's complex networks. Instead of trying to formulate a static set of plans, an information architecture tries to plan for change. It is recognized that there will be many different information systems and types of computers. The architecture tries to provide the framework that allows these various environments to function together as an integrated whole.

Part I

The User Interface



```
I N G R E S                                         November 05, 1988
                                                    1:21:26 pm
   Employee Task Assignments               11/05/88 13:21
     Name  Alcott, Scott
        Title      Sr Programmer            Hourly Rate
        Manager    Wolfe, Neal               $ 50.00
        Task Assignments
        Project                     Task            Hours
        Advertise                   Design              8
        Advertise                   Implement           5
                                      Total Hours      13
                                      Total Cost  $ 650.00
 Next(Enter)  Recalc(2)  DeleteEmp(3)  RemoveTask(4)  >
```

Overview

Part I of this book discusses what the user sees. We assume for the time being that back end and the network are black boxes—the application asks for data and they magically appear. We also assume that a database is already in place; issues of how to design a database are deferred to later in the book.

Chapter 2 starts with a discussion of the general-purpose user interface. This is a tool that allows the user, without any programming or application development, to access data in the database. General-purpose tools allow the user to browse and update data, to format reports, and to generate graphs.

Chapter 3 discusses the tools available for building applications. Instead of using a general-purpose user interface, the application developer is able to customize the INGRES user interface for specific projects and tasks. This customization can be quite simple or can involve large programming projects.

Chapter 4 deals with how the user-interface tools are changing as the platforms that they run on change. As users move from a terminal-oriented environment to workstations with bit-mapped graphics screens, a wider range of options is available to design more powerful and more intuitive user interfaces. Two projects are discussed, one a commercial software product, the other a university-based research project, that provide solutions for workstations.

Chapter

2

General-Purpose User Interfaces

Forms-Based Interfaces

Most of the INGRES subsystems use a forms-based interface, which means that the terminal screen is broken up into several different data display areas. Each of these areas is a field. The collection of fields makes up a form. All of the figures in this chapter are examples of forms.

The most obvious use of a form is for data entry applications. In a data entry application, the user fills in each field, then uses the TAB key to move to the next field. If a mistake has been made, the user can move back to the field that is in error and fill in new values.

Associated with each form is a menu. The menu presents the user with a series of actions that can be taken after the form is filled out. In the data entry application, the menu might have options to save the data, clear the form, and exit the application.

A single form is often used for different purposes. For example, a form might be used in one application for data entry, and then subsequently used by another user to browse data. Each of these uses would have a different menu associated with it. The combination of a form and a menu is known as a frame in INGRES.

Each field on a form has different characteristics associated with it. These characteristics include visual characteristics and a variety of hidden characteristics known as attributes.

Visual characteristics can include highlighting the area of the field, which makes the location of different fields more apparent to the user. It is also possible to have fields blinking, underlined, or boxed. A blinking field might be used in a security application to indicate that certain pieces of data are confidential. The next chapter discusses how a programmer might make a field change from reverse video (highlighting) to blinking when confidential data are retrieved from the database.

Other attributes govern the use of a field, for example, a field might be declared mandatory. If a user attempts to tab over this field without entering any information, an error message will appear at the bottom of the screen.

It is also possible to automatically convert the value of a field to upper- or lowercase to ensure the consistency of data. Another consistency check is the validation criteria associated with a field. An example of a validation check is declaring that the "state" field for an address must contain a valid state abbreviation. The section of this chapter on VIFRED, the Visual-Forms-Editor, shows how a user can establish a variety of different criteria associated with fields on a form.

A powerful feature of INGRES is that it is not necessary to design a form to use INGRES. Most of the subsystems, such as the QBF utility, discussed next, are able to build a default form. Later on, VIFRED can be used to make the form more sophisticated.

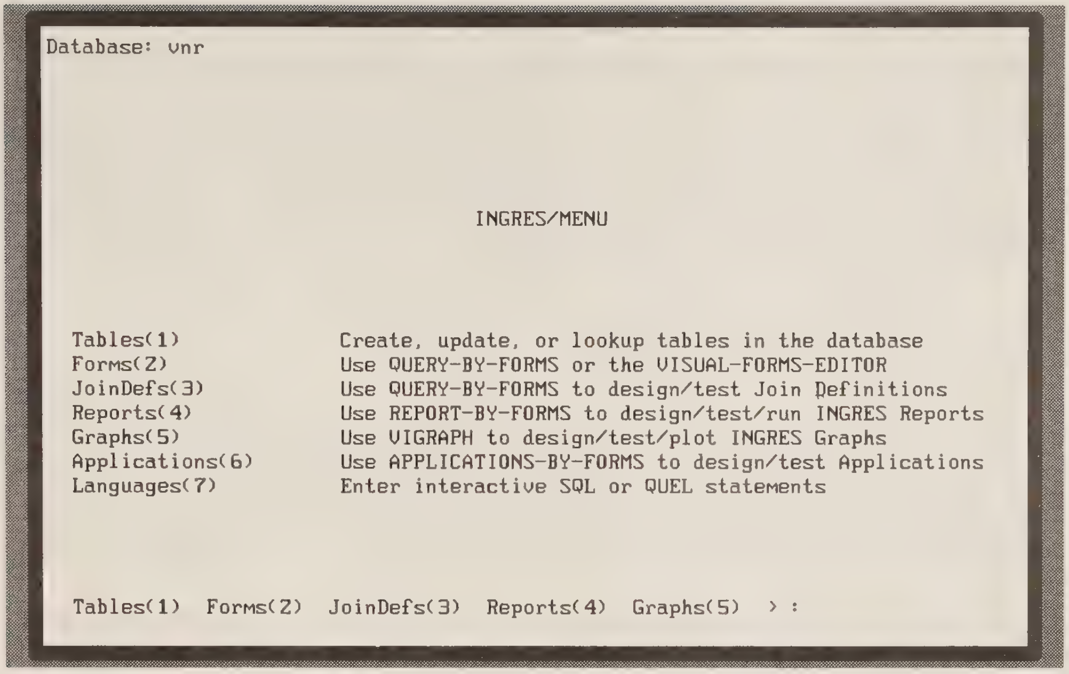
The first general-purpose user interface discussed in this chapter is QBF, which can be used to browse and change data in a database. Since no programming is needed, QBF can be used by a fairly wide variety of users. After QBF, this chapter discusses the Report-By-Forms (RBF) utility, which is used to build a report from the database. By contrast, QBF is used for interactive browsing of data rather than to create the more formal printed reports produced by the reporting subsystems.

Next, the Visual-Graphics-Editor (VIGRAPH) subsystem is discussed. This extends the forms concept to include a new type of object—the graph. Like QBF or RBF, VIGRAPH is a visually oriented system that allows nonprogrammers to quickly design graphs to display data in the database.

The last forms-based subsystem discussed in this chapter is VIFRED—the Visual-Forms-Editor. VIFRED can be used in conjunction with QBF to produce a more sophisticated user interface. Forms, however, play a more important role than just providing a more sophisticated version of QBF; they are used in a wide variety of applications. Chapter 4 discusses how a form that was developed for use in QBF—a general-purpose user interface—can then become part of custom interfaces developed in the Application-By-Forms (ABF) subsystem.

This chapter then discusses two INGRES subsystems that are not forms based. The Report Writer is a command language version of RBF, which requires the user to write a small program, the report specification. The Report Writer is used to produce more sophisticated reports than are possible in RBF.

Finally, the terminal monitors are discussed. Terminal monitors are subsystems that allow users to directly enter SQL or QUEL commands and have them executed. QBF is an example of a forms-based user interface that generates SQL commands for the user, who never sees the SQL commands. Certain users will need to directly execute SQL commands. In particular, application developers will want to test SQL statements before using them in programs such as ABF applications. The terminal monitor allows the



```
Database: vnr



                               INGRES/MENU



 Tables(1)            Create, update, or lookup tables in the database
 Forms(2)             Use QUERY-BY-FORMS or the VISUAL-FORMS-EDITOR
 JoinDefs(3)          Use QUERY-BY-FORMS to design/test Join Definitions
 Reports(4)           Use REPORT-BY-FORMS to design/test/run INGRES Reports
 Graphs(5)            Use VIGRAPH to design/test/plot INGRES Graphs
 Applications(6)      Use APPLICATIONS-BY-FORMS to design/test Applications
 Languages(7)         Enter interactive SQL or QUEL statements



 Tables(1)  Forms(2)  JoinDefs(3)  Reports(4)  Graphs(5)  > :
```

Courtesy of Relational Technology

Fig. 2-1 INGRES/MENU Main Screen

application developer to quickly see the results of an SQL statement before incorporating them in an application.

INGRES/MENU—Accessing the Subsystems

To access INGRES data using forms-based interfaces, the user would simply type *INGMENU* and the name of the database to be accessed. The user would then see the screen shown in Figure 2-1. INGRES/MENU provides a single access point for all the different subsystems available in INGRES. Although it is possible to access the subsystems from the command line, it is easier to train users to use INGRES/MENU. The operation is quite simple—the user picks a menu option. The screen display is a form with no fields on it (at least no fields that the user can fill in; there is a display-only field on this form).

The *TABLES* menu option is a simple way to examine the different tables in the database. The user is presented with a catalog of tables and can ask for information on a particular table. This information includes the number of rows in the table, a list of the columns in the table, their data types, and information on the physical structure of the table such as the storage structure and keys.

The tables utility is an example of a data dictionary; it examines the definitions of data. QBF lets a user examine the contents of tables, while the tables utility looks at their definition.

The information examined in the tables utility is known as meta-data, data about data. Throughout this book a variety of forms of meta-data will be seen. Usually, these data will appear as a type of catalog. Catalogs of forms, reports, and other objects in the INGRES database are often displayed. Part IV of this book looks more extensively at data dictionaries and their use for the internal management of a database system, as well as the for storing the definition of objects in a computer-aided software engineering (CASE) environment.

Forms and Terminal Independence

One important characteristic of forms is that they are terminal-independent; INGRES allows the same form to be displayed on a wide variety of different terminals. When the form is displayed, INGRES maps the generic definition of a form into the low-level commands necessary to display the form, clear the screen, do reverse video, and other operations.

The definition of a terminal is contained in two special files. First, there is a terminal capability (termcap) file that contains the low-level commands necessary to interpret incoming information from a keyboard and to perform operations on the screen. The second file is a mapping file, which takes generic functions (such as move forward) and maps them to specific keys (such as the tab key). A user can modify the mapping files to change the keys that execute various functions.

Changing mapping files is useful when INGRES is brought into an environment that already has users trained to look for certain keys. If users are using DEC's ALL-IN-1 office automation environment, for example, it would be sensible to map the INGRES keys to the equivalent keys used to perform functions in ALL-IN-1. In this case, the help function would be mapped to the PF2 key on a VT-style keyboard.

The flexibility of mapping files and termcap descriptions helps make INGRES a portable environment. A form can be developed on a PC and later moved to VAX without changing the definition of the form. Portability allows applications to be developed in one environment without later changing the code when the application is moved to another environment.

Query-By-Forms

QBF is a general-purpose user interface that provides all the functionality that is needed for many database applications. QBF allows users to enter, retrieve, and update data. Tables in the database are listed in a catalog; the user simply selects the appropriate table by pointing to it and then performs the desired operations on that query target.

It should be noted that a database may not let a particular user use QBF to look at or change data. The database may have certain security constraints placed on the data that limit access. In this chapter, we don't worry about those back-end constraints; we assume the user is able to access data. In the next part of this book, we will stop looking at the back end as a "black box" and start examining exactly who is allowed access to what data (and how quickly they can access them).

It is possible in QBF to operate on several different database tables at once. A sophisticated form of QBF, known as join definitions, allows special rules to be set up that govern how data are entered into multiple tables from a single form. We start first with the simpler, single-table form of QBF and then move on to join definitions.

Simple QBF Operation

The basic QBF operation consists of choosing a table and performing one of three operations on it:

- retrieve data
- append data
- update data

To choose a table, the user is presented with a list of tables in the database, known as a catalog. Later on, we will see that other catalogs exist for forms, reports, and other objects that are stored in the database. Figure 2-2 shows an example of a tables catalog. The user uses the arrow keys on the keyboard to select a query target.

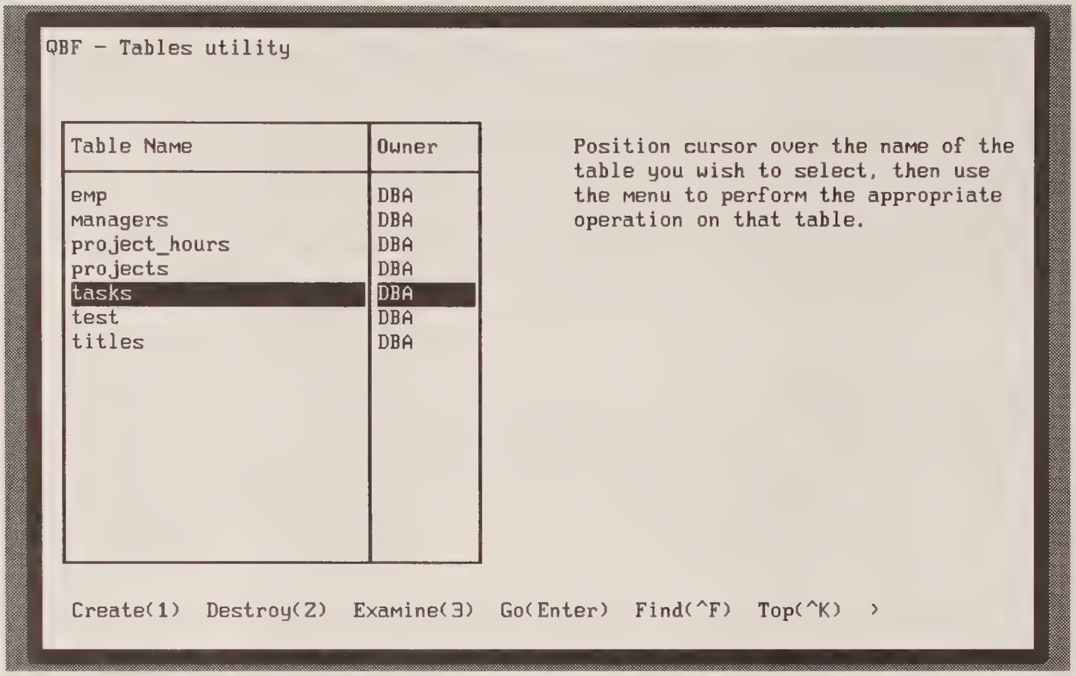
After choosing the table, the user chooses what type of form they would like to use. The form can have simple fields or table fields. A simple field means that a single row of data can be displayed at one time. A table field can display several rows of data at once.

In the basic retrieve operation, the user first sees a blank form. The user then fills in values on the fields that specify which rows of data the user wishes to see (see Fig. 2-3). The simplest query involves no specification—this means the user wants to see all rows of data.

By tabbing to a specific field, the user can qualify the query, which is equivalent to the "where" clause on an SQL select statement. For example, filling in *B** on the name field of the form requests all rows in the tasks table that begin with a "B" followed by any other character. Alternatively, the user could have put the value *Bottorff* in the field, which would retrieve all names beginning with the characters "Bottorff."

It is possible that a single row of data meets this criterion. However, it is also possible that there are many rows of data that meet the search criterion. QBF displays the first row of data that meets the qualification *NAME = "B*"*, then displays a new menu on the screen (see Fig. 2-4).

If the form has a series of simple fields, this new menu lets the user pick the next row of data. By repeatedly picking NEXT, the user is able to examine each row that meets the selection criterion. After the last row is displayed, the user will see a message "no more rows in query."



QBF - Tables utility

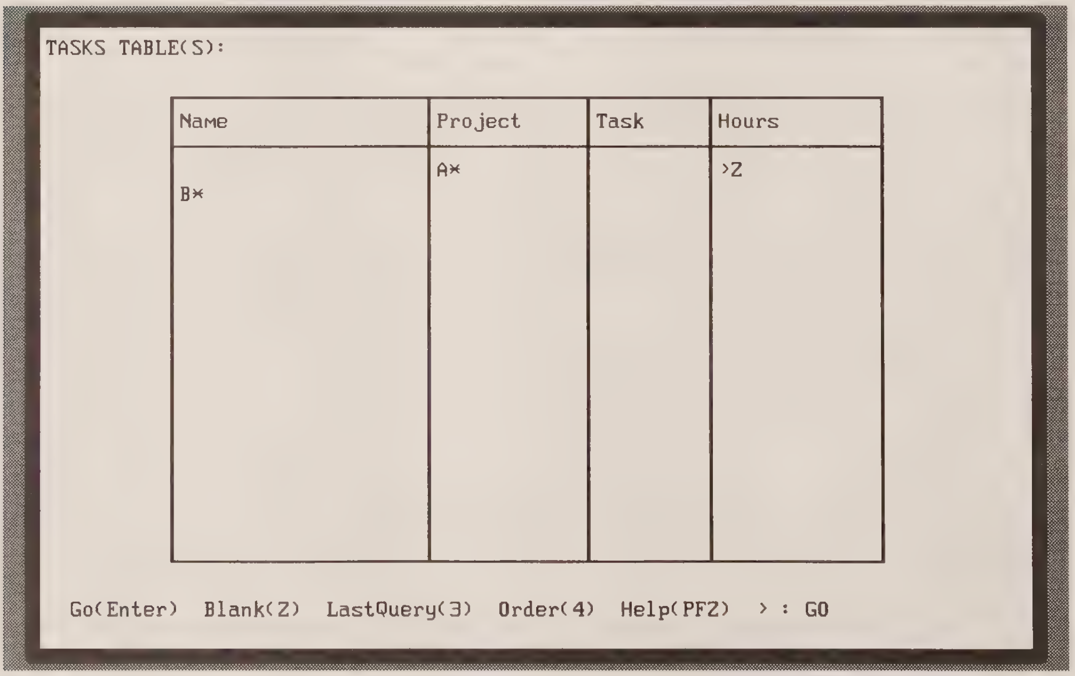
Table Name	Owner
emp	DBA
managers	DBA
project_hours	DBA
projects	DBA
tasks	DBA
test	DBA
titles	DBA

Position cursor over the name of the table you wish to select, then use the menu to perform the appropriate operation on that table.

Create(1) Destroy(2) Examine(3) Go(Enter) Find(^F) Top(^K) >

Courtesy of Relational Technology

Fig. 2-2 Picking a QBF Query Target

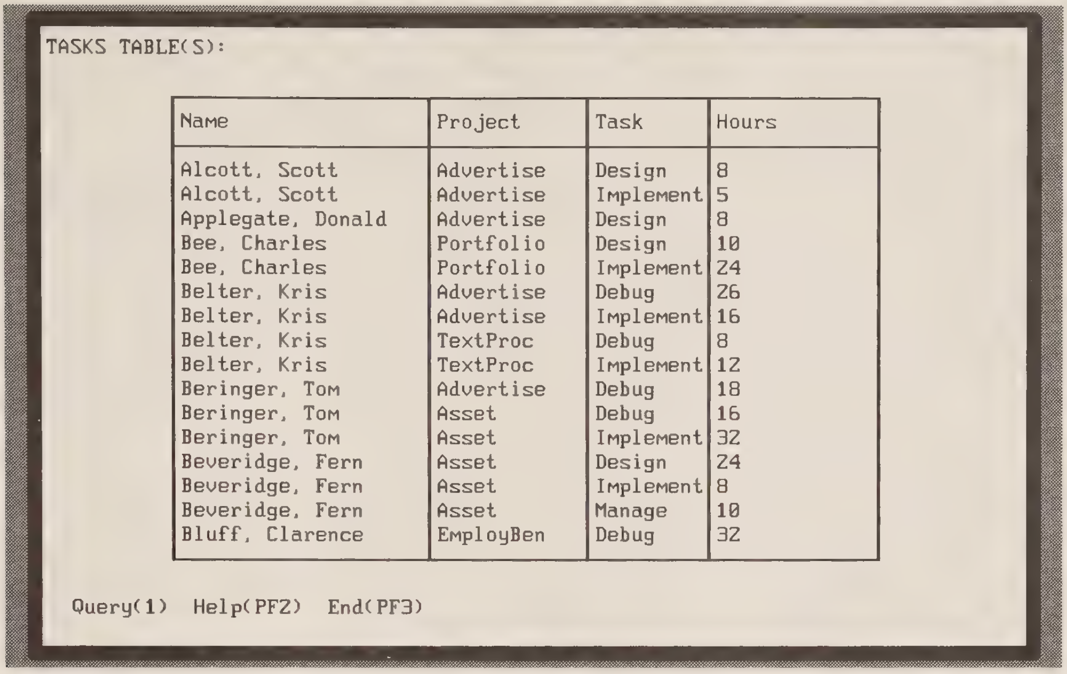


TASKS TABLE(S):

Name	Project	Task	Hours
	A*		>Z
B*			

Courtesy of Relational Technology

Fig. 2-3 Search Criteria for a QBF Retrieval



TASKS TABLE(S):

Name	Project	Task	Hours
Alcott, Scott	Advertise	Design	8
Alcott, Scott	Advertise	Implement	5
Applegate, Donald	Advertise	Design	8
Bee, Charles	Portfolio	Design	10
Bee, Charles	Portfolio	Implement	24
Belter, Kris	Advertise	Debug	26
Belter, Kris	Advertise	Implement	16
Belter, Kris	TextProc	Debug	8
Belter, Kris	TextProc	Implement	12
Beringer, Tom	Advertise	Debug	18
Beringer, Tom	Asset	Debug	16
Beringer, Tom	Asset	Implement	32
Beveridge, Fern	Asset	Design	24
Beveridge, Fern	Asset	Implement	8
Beveridge, Fern	Asset	Manage	10
Bluff, Clarence	EmployBen	Debug	32

Courtesy of Relational Technology

Fig. 2-4 Results of QBF Search

In the case of a table field, there is no need for a "NEXT" menu option. This is because the table field displays several rows of data. If all the rows do not fit on the display, the user simply goes to the last row of the table field and uses the arrow key to examine the rest of the retrieved rows.

At any time during this process, a user can pick a QUERY menu option. This option clears the screen and lets the user specify a new set of selection criteria. Often, the first query (i.e., NAME = B*) is too broad. By filling in values in multiple fields, the user can narrow the search. For example, the user could fill in the project field with the value "A*."

In Figure 2-3, the values A* and B* are on different rows of the table field. This is telling QBF that the user wants to see rows with either projects beginning with an A or names beginning with a B. If both values were on the same row of the table field, the user would be requesting all rows of the table that have a project beginning with an A and a name beginning with a B.

More complicated forms of selection criteria are also available. For numeric fields, the user can specify a query operator and a value. Normally, the query operator is "=." By filling in a 2 in the hours field, a user is asking QBF to retrieve all rows with *hours = 2*. Other query operators let the user pick values greater than, less than, or not equal to a certain value. Filling in ">2" on the hours field is equivalent to requesting all rows in the tasks table with more than 2 hours. Filling in ">=2" asks for all rows with two or more in the hours field.

For text fields, wild cards are used to broaden the search criteria. Filling in M* in the name field requests all names in the tasks table starting with M. The "*" means any set of characters. Filling in *M asks for all names ending with an M.

A related wild card character is the "?," which matches a single character. If the user thinks that a name may have been spelled incorrectly in the database, she might specify a query for *M?STERS*. This would retrieve the names MASTERS and MESTERS, but not the name MEASTERS. To get MEASTERS the user would have to use the wild card character that matches several characters (M*STERS).

One problem with the query M* is that users may have entered names without capitalizing the first letter. The query M* requests any names beginning with a capital M. To request a name beginning with either a capital M or a lowercase m, the query would have to be specified as "[Mm]*." This form requests a name beginning with any of the characters in brackets, followed, in this case, by any string of characters.

Note that "*" and "?" as wild card characters do not conform to a strict definition of SQL. The SQL equivalents are the "%" and "_" characters. Since most users are familiar with the asterisk as a wild card character, Relational Technology chose to support both SQL and more traditional semantics for these functions.

The basic QBF retrieval thus consists of filling in search criteria and cycling through the rows in the table that meet these criteria. The retrieval can be interrupted at any time and a new query formulated. When users pick a table, they are first asked if they wish to see the form as simple fields or table fields. In the table field, several columns are grouped together, and several rows of data can be displayed. Instead of picking the next NEXT option to see each row, several rows are displayed simultaneously.

It is possible that a user specifies a query that would retrieve more rows than can be displayed in the table field. For example, *name=M** might select 34 rows of data. The table field may only have 10 rows in it. QBF keeps the excess rows in an invisible buffer, called the data set. When the user positions the cursor on the last row of the data set and then presses a down arrow, the table field scrolls up displaying the next row in the data set.

Arrow keys and the scroll keys (e.g., page up and page down) are thus used to scroll up and down within a table field. If no more rows are available, a message will be shown to the user. The tab key is used to move from one column to another. It is possible that a form has a combination of simple and table fields. The tab key would be used to move in between fields and columns. Once on a table field, the arrow keys would be used to scroll up and down the table field.

Table fields are a convenient way of specifying complex queries. In Figure 2-3, the user is asking for two different types of rows from the database. either rows that meet the criteria in row 1 of the table field or rows that meet the criteria in row 2 of the table field.

In addition to retrieving information from the database, the user can append and update data. An append operation consists of a single step—a blank form followed by an append operation. An update operation consists of two steps. First, the user has to specify what information she wishes to see. This is equivalent to a retrieve operation.

Then, for each of the rows retrieved, the user can update or delete information. Each subsequent row is then examined until the "NO MORE ROWS IN QUERY" message is displayed.

Join Definitions

Often, the data to be examined by a user exist in several different tables in the database. The tasks table, for example, has a description of people assigned to a particular task. Another table, the employee table, may have information on the individuals in the organization.

With two tables in the database, it is possible to execute QBF twice, once for each row. This is not a very useful technique for keeping track of both projects and people assigned to those projects. Instead, most people will want to look at both tables simultaneously.

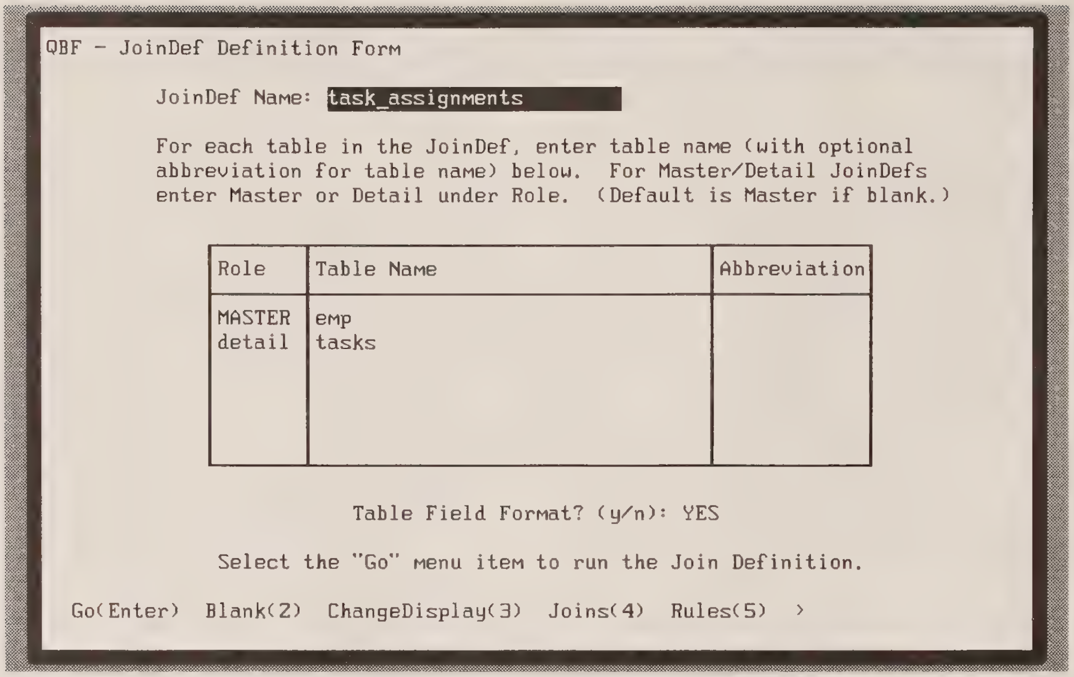
In SQL, the user would join the two tables together. A join tells the database to combine the two tables, using the values in one or several fields to match up rows in the two tables. In this case, all rows in the emp table would be combined with their equivalent tasks in the tasks table, using the employee name field as the match field.

There are two ways of looking at this join. First, the user might wish to see every person, and all tasks that the person is working on. It is possible that a person is on several projects. This is known as a master-detail query. The tasks table would be the master table. For each master, that is, each row in the tasks table, there may be several detail rows, that is, rows from the emp table. Remember that there are also several master rows, each one with several detail rows associated with it.

Another form of master-detail query would make the task the master. For each task, it is possible to have several people assigned to it. Since QBF does not know exactly how a particular user wants to look at the two tables, it is necessary to create a new type of object, called the join definition. A join definition is composed of several tables in the database and some rules on how to put those tables together. Figure 2-5 shows a join definition in QBF.

Join definitions (JoinDefs), like forms and tables, are stored in the database. Upon starting QBF, the user is first asked what type of query target he wants to look at. A table is one example of a query target; in this case the user is shown a catalog of available tables. A JoinDef is another type of query target, and, again, a catalog of JoinDefs is presented to the user.

Instead of using an existing JoinDef, the user can formulate a new one. The first task is to specify which tables are masters and which tables are detail tables. It is also possible to eliminate certain columns from the JoinDef; perhaps the employee's manager is of no interest for this particular application. Eliminating the manager means that less data are shown on the screen (not to mention the therapeutic value to the employee who gets to design the form). Figure 2-6 shows the screen in QBF that allows the user to specify which fields are to be displayed, as well as to change how tables are joined.



QBF - JoinDef Definition Form

JoinDef Name: task_assignments

For each table in the JoinDef, enter table name (with optional abbreviation for table name) below. For Master/Detail JoinDefs enter Master or Detail under Role. (Default is Master if blank.)

Role	Table Name	Abbreviation
MASTER	emp	
detail	tasks	

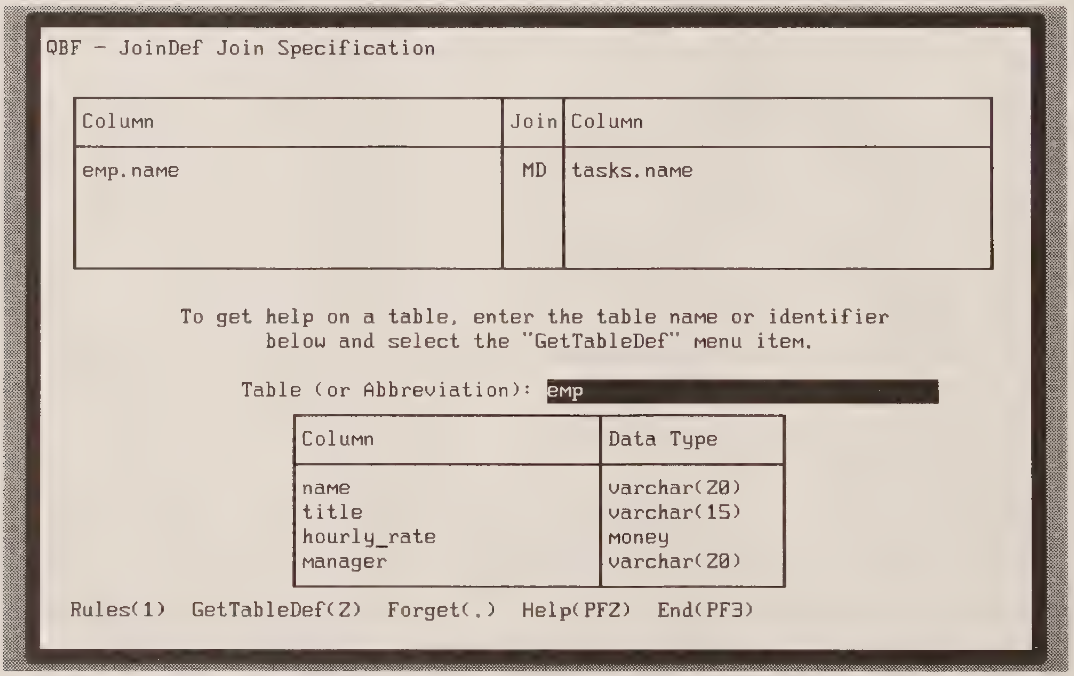
Table Field Format? (y/n): YES

Select the "Go" menu item to run the Join Definition.

Go(Enter) Blank(2) ChangeDisplay(3) Joins(4) Rules(5) >

Courtesy of Relational Technology

Fig. 2-5 Master and Detail Tables for a Joindef



QBF - JoinDef Join Specification

Column	Join	Column
emp.name	MD	tasks.name

To get help on a table, enter the table name or identifier below and select the "GetTableDef" menu item.

Table (or Abbreviation): emp

Column	Data Type
name	varchar(20)
title	varchar(15)
hourly_rate	money
manager	varchar(20)

Courtesy of Relational Technology

Fig. 2-6 Specifying Which Columns Are in a Joindef

In addition to specifying the master-detail relationship of the tables, it is sometimes necessary to specify delete and update rules for the JoinDef. When a user is doing an update operation in QBF and picks the delete menu option, the operation is unambiguous on a single table query target. Delete means delete the current row.

In a JoinDef, if the user says delete, the question if whether to delete the master or the detail table. If employee is the master and tasks is the detail, there are three possible interpretations of a delete operation. Delete could mean delete the detail row—eliminate this particular task. It could also mean delete the master—remove the employee from the database. Finally, it could mean delete both rows—eliminate both the task and the people assigned to that project.

The particular operation needed depends on the application. In this example, a user might define three different JoinDefs, one for each type of operation. The user would then pick the JoinDef appropriate to that particular operation. Note that the screen would look the same in all cases; it is only the operation on the database that changes.

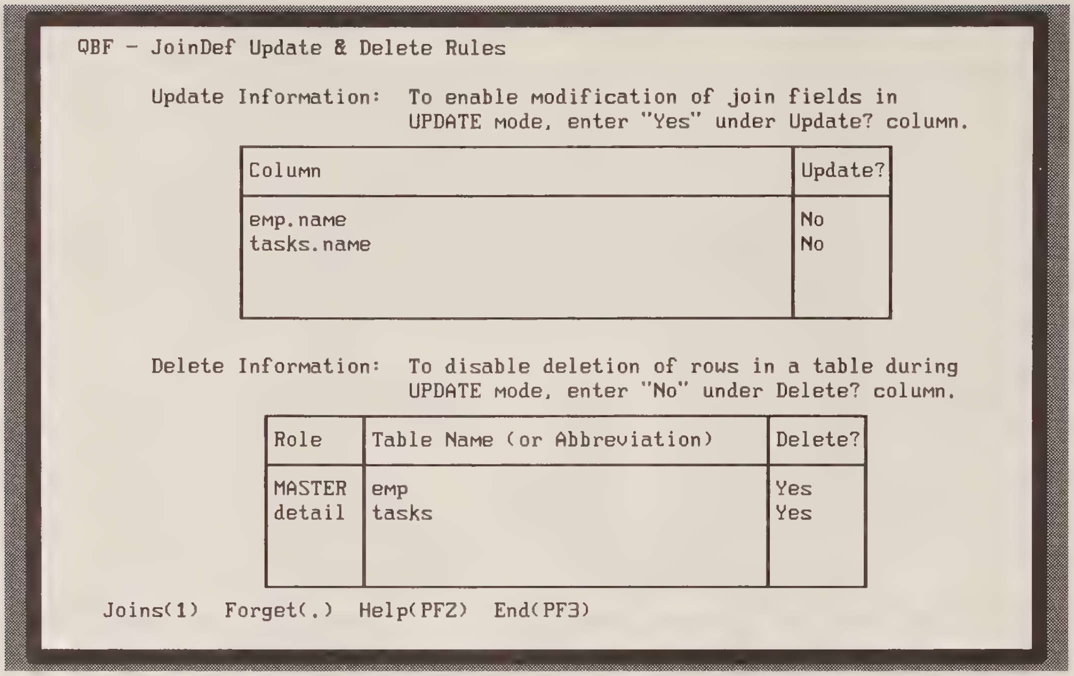
Update rules specify a similar type of information for the JoinDef. When a joined column, in this case the name column, is updated, the question is whether to update the master or the detail version of the name (or both). If an employee gets married, both should be updated. If a person is reassigned to a new project, only one of the values should be changed. Figure 2-7 illustrates update and delete rules for a JoinDef.

A JoinDef thus consists of two types of information. First, the tables that are joined together and the columns used to perform the join is specified. Tables are designed as either master or detail tables. Second, update and delete rules are specified. Once the JoinDef is formulated, it becomes part of the catalogs as an available query target. The users of the JoinDef operate QBF in the same way as they would on a single table. The only exception is the *nextmaster* menu option, which picks the next master row from the database (see Fig. 2-8).

The advantage of QBF, either the single table or the JoinDef version, is quite simple—users can browse and change tables in the database without any programming. Once users are taught how to use QBF, they are able to access data without any further assistance from the database administrator. QBF is thus an end-user tool—it requires no programming and is easy to learn.

There are times when QBF by itself is not appropriate. First, some users may not be able to master QBF. For these users, a simpler custom application may be necessary. This application would have operations customized to specific tasks, such as adding a new personnel record. The second situation is where the underlying database has a complex structure. Users may not be able to interpret the underlying tables and would be unable to use QBF without further assistance. Sometimes, the database administrator can define a series of JoinDefs or views to simplify the structure of the database.

Related to this is the problem of referential integrity. Referential integrity defines relationships between different tables in the database. To illustrate this problem, consider a purchase order application. There may be one table for outstanding purchase orders and another for approved vendors. A typical policy in this situation is that new purchase orders are not entered without the vendor being on the approved vendors list.



QBF - JoinDef Update & Delete Rules

Update Information: To enable modification of join fields in UPDATE mode, enter "Yes" under Update? column.

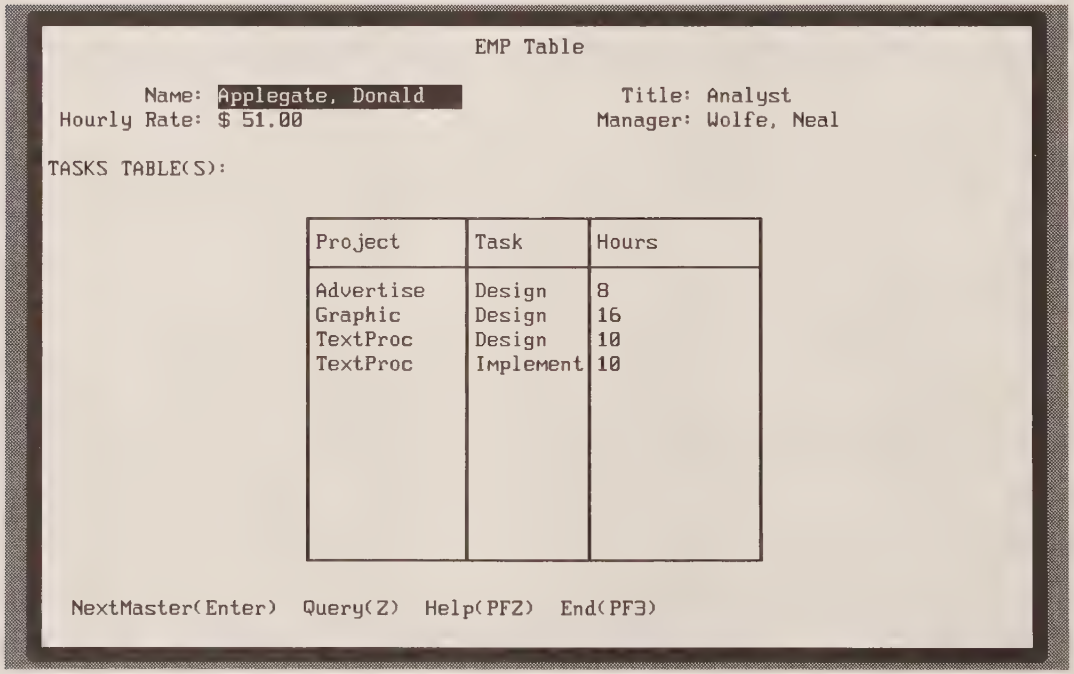
Column	Update?
emp.name	No
tasks.name	No

Delete Information: To disable deletion of rows in a table during UPDATE mode, enter "No" under Delete? column.

Role	Table Name (or Abbreviation)	Delete?
MASTER	emp	Yes
detail	tasks	Yes

Courtesy of Relational Technology

Fig. 2-7 Update and Delete Rules for a Joindef



EMP Table

Name: Applegate, Donald

Hourly Rate: $ 51.00

Title: Analyst

Manager: Wolfe, Neal

TASKS TABLE(S):

Project	Task	Hours
Advertise	Design	8
Graphic	Design	16
TextProc	Design	10
TextProc	Implement	10

Courtesy of Relational Technology

Fig. 2-8 Using a Joindef in a QBF Retrieval

Likewise, no vendors are deleted from the approved vendors list if there are any purchase orders outstanding. The integrity of one table is defined in reference to another.

The problem with QBF in this particular situation is that it is quite simple to append a row to the purchase orders table, bypassing the policy of the company. In this situation, a custom application that first checks that the appropriate policies are met and then performs the update would be more appropriate.

We will look at two methods that can accomplish this goal of building integrity or validation into the front-end application. First, VIFRED allows some simple rules to be formulated that address some of these problems. Second, the INGRES 4GL allows more complex integrity constraints to be designed that complement the facilities in VIFRED.

VIFRED requires more work than straight QBF operations, although the amount of work is actually quite small and can be performed by the end user. ABF and INGRES 4GL require even more work. If QBF, by itself, does the job, it makes sense to use this tool. If the next level of sophistication is needed, then VIFRED can be used to ensure a higher level of integrity. Finally, if both these tools are inappropriate, the 4GL can be used.

An important attribute of INGRES is that these different levels of sophistication are all available within the same general framework. The tools that users learn in QBF can all be applied to VIFRED and ABF. It is possible for users to move up to higher levels of sophistication without discarding the skills that they learned with the simpler tools.

Report-By-Forms

QBF is an interactive tool—it allows the user to browse data on the screen. Often, a more formal report is needed with more structure than QBF can provide. For example, we might wish to print each project on a separate page, with appropriate summary information for each project at the bottom of each page. This type of output is known as a report. Reports can be printed or can be viewed on the screen. Like tables, forms, and JoinDefs, reports specifications are also stored in the database. To run a report, the user is presented with a catalog and chooses the appropriate report.

To define a report, there are two different methods available. First, the user can use RBF. This is similar to using QBF to formulate a join definition. In both cases, forms are used to specify the exact nature of the operation.

An alternative to RBF is the Report Writer. This is a command language interface—the user writes a program that specifies how the report behaves. Obviously, RBF is easier to use and requires less work because no programming is involved. If RBF can produce an appropriate report, it is the appropriate tool to use. Only when the limits of RBF have been reached does the user need to look at the Report Writer.

With RBF, as with QBF, report definition begins by selecting data—a query target. RBF will set up a default report for a view or a table. A view is actually an SQL statement that is executed, but the user sees only the resulting table. It is somewhat like a JoinDef, except that the JoinDef also includes update and delete rules. The view goes further than a JoinDef in the ability to use the full power of SQL to select data, instead of just specifying the columns of data needed (the where clause is used to select particular rows). It is possible, indeed common, to use views as part of a JoinDef to combine the power of both data combination methods in a single QBF operation.

An advantage of views in the report environment is that they can be used to create calculated columns and other information that the RBF cannot produce. To the user, these columns look like simple database tables, but the aggregate and calculated columns are actually rederived each time the view is accessed. For example, a simple view definition might be:

```
create view sales_total
                ( item, quantity, unit_price, total_cost)

as
                select i.item, i.quantity,i.price,
                           i.quantity*i.price
                from inventory i
```

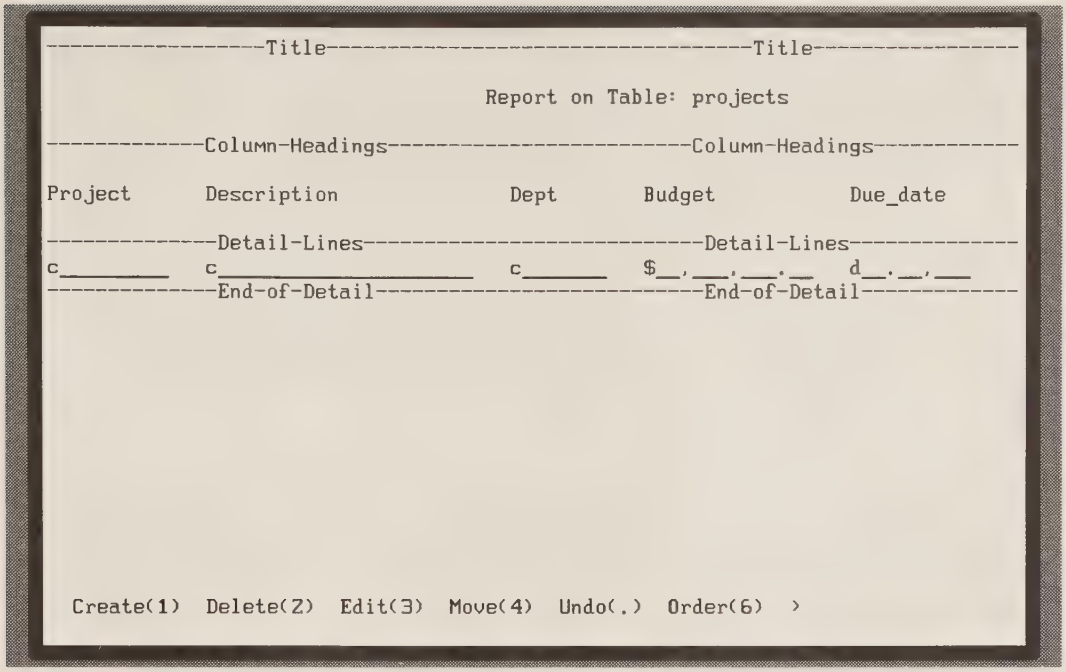
This view creates a virtual table that includes not only the quantity and unit price for a particular sale, but the total cost of the sale. The base information is already in the database but is not in a form easily read by users. If someone creates this view, every time the "table" sales_total is accessed, a select statement is issued to the inventory table.

Creating views is a convenient way of combining database information into virtual tables that simplify the structure of the underlying database for certain classes of users. Once the view is created, those users can then use RBF to write reports without having to know the details of the underlying structure of the database or how to use SQL to create aggregate information and calculated columns.

A default RBF report includes a title, column headings, and data (see Fig. 2-9). The user then edits this report to include more sophisticated information. Typically, this begins by translating the title and column headings into their English, French, Kanjii, or Spanish equivalents. The title, column headings, and data display areas can also be moved to new locations on the screen.

This basic report layout is a template for the report when it is run. When the report is actually run, data are substituted for the format information that appears in the initial layout screen. RBF is thus a visual report writer—the user moves the pieces of the report where they should appear on the output.

The data in the report originally appears as it is stored in the database. Thus, a simple integer will appear on the layout screen as *i*_ _ _ _ _ _ _ _ (i9), signifying that the number will appear with nine columns.



```
-----------------Title------------------------------------Title----------------

                                   Report on Table: projects

------------Column-Headings-------------------------Column-Headings------------

Project      Description              Dept       Budget           Due_date

-------------Detail-Lines---------------------------Detail-Lines--------------
c_________   c_____________________   c_______   $__,___,___.__   d__.__,___
-------------End-of-Detail--------------------------End-of-Detail-------------








  Create(1)  Delete(2)  Edit(3)  Move(4)  Undo(.)  Order(6)  >
```

Courtesy of Relational Technology

Fig. 2-9 Layout of an RBF Report

Editing the data allows the length and the display format of information to be changed. For example, an i8 field (an eight-place number) can be changed to i2. A long character field can be shortened to reflect the actual length of the data in the database. If the display format is too short, the data will be truncated when retrieved from the database. Sometimes a deliberate decision is made to view only the first few characters of a field, as in the case of viewing the first 10 characters of a project description.

Often, data are stored as numbers in the database to speed their retrieval. A common example is storing a social security number as an integer to make it quicker to perform sorts by that item. When displaying data it makes more sense to put the dashes back into the display. RBF allows numeric templates to be defined that, specify exactly how data are to be displayed.

A numeric template uses example characters to specify how data should be displayed. The character N, for example, signifies that a number should be placed in this column, and if there is no datum, a zero should be placed in that column. This is frequently used for money columns:

"NNNNNNN.NN"

This template would output the data 234567.1 as

0234567.10

More complicated templates allow commas to be placed in the appropriate position, just as the previous template had a period in it. A z character says to print the number if it exists; otherwise print a blank. Z characters are thus used to the left of the decimal point, N characters to the right, for a typical money numeric template:

Template: "Z,ZZZ,ZZZ.NN"

Data in Table	Data Displayed
234567.1	234,567.10
1234567	1,234,567.00

Templates are also available for date formats. These formats specify exactly how dates in the database should be displayed. The default date format includes all date information, including time down to the second. Often, only the day, month, and year are needed. The template for this combination would be:

d"February 3, 1901"

This template causes the full name of the month to be printed out. If the template had said "Feb." instead, the abbreviation for the month would have been printed.

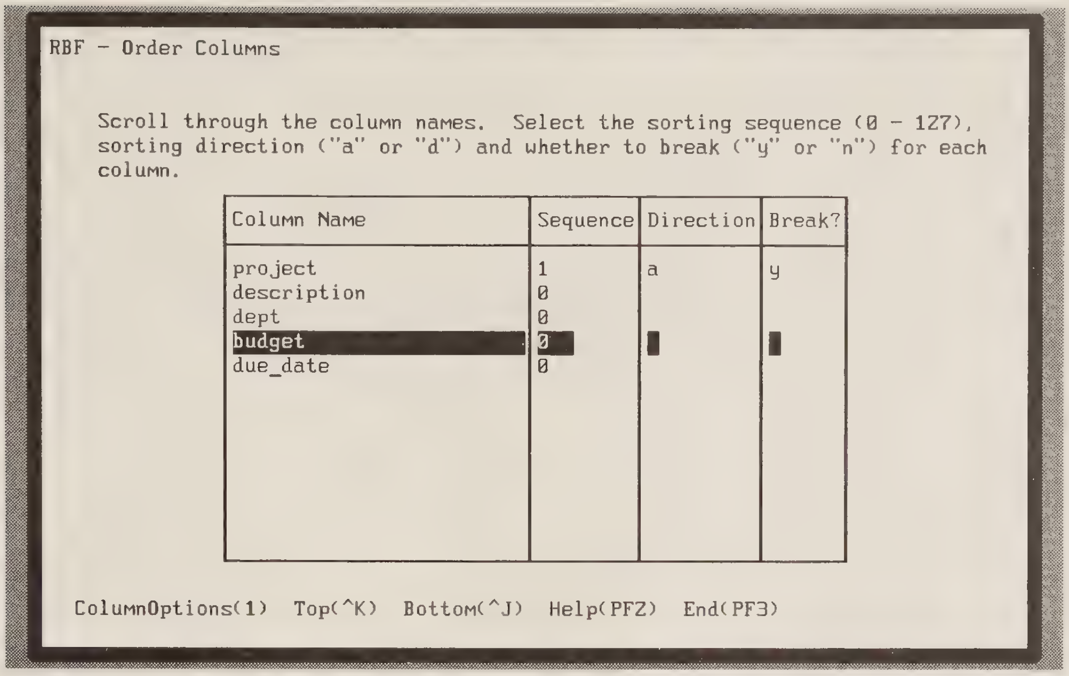
Once the generic appearance of the data is defined, the user defines the sort order of information (see Fig. 2-10). For the project table, the data are being sorted by project name in ascending order. When the sort is specified, it is possible to specify special operations on the columns that are part of the sorting. These are known as break columns. Whenever a new value of the sorted column occurs, three types of actions can occur:

- control over the appearance of repeating values
- control over the spatial appearance of the report
- definition of aggregates

Whenever a report is sorted by project, one can assume that there will be several rows of data for that project. The default operation of a report is to print all columns for all rows of data. This means that the same project name will be repeated for every department assigned to the project.

The user is able to define when to print repeating values for a break column. First, the columns can be only printed when the value changes—when a new project occurs. This is fine if projects always fit on one page. However, if the project has many values, it is possible that it will stretch to the next page. To handle this situation, it is common to have the value for the break column printed at the top of a new page, whether or not the value has changed.

The second attribute of a sorted column is the ability to control the appearance of the



RBF - Order Columns

Scroll through the column names. Select the sorting sequence (0 - 127), sorting direction ("a" or "d") and whether to break ("y" or "n") for each column.

Column Name	Sequence	Direction	Break?
project	1	a	y
description	0		
dept	0		
budget	0		
due_date	0		

Courtesy of Relational Technology

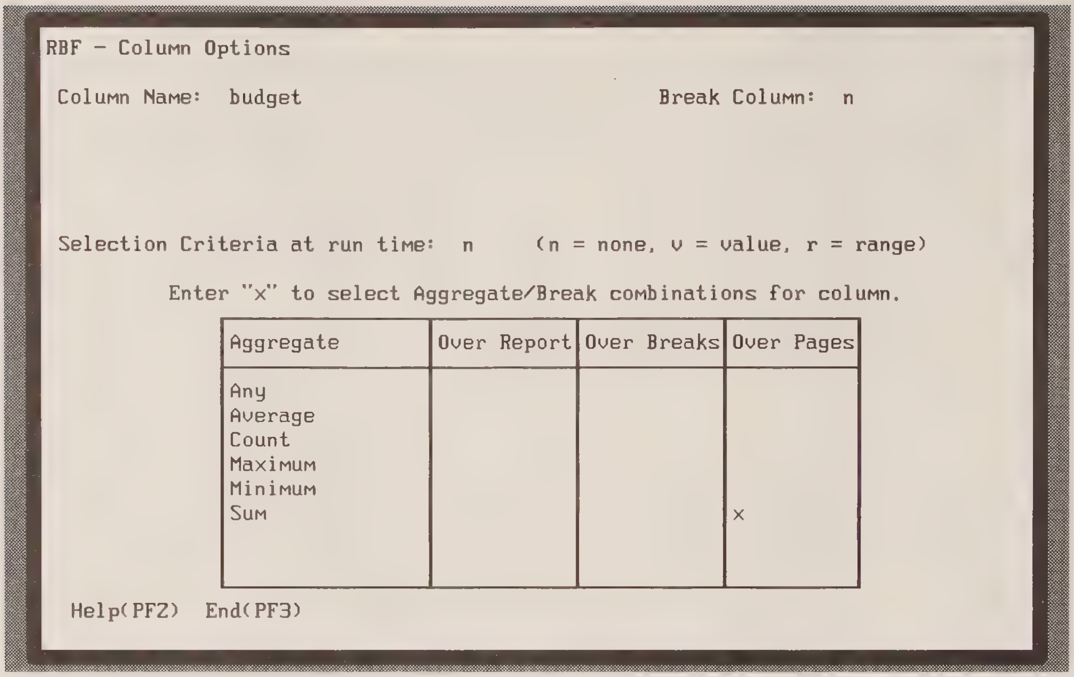
Fig. 2-10 Ordering Columns

report when values change. Normally, the next row of data will be printed on the next line of the report. The user can decide to skip a certain number of lines after the value changes, or to skip to the next page of the report.

The last attribute of a column is the ability to provide aggregate information. Whenever the last row of a project is reached, the user may wish to print the total budget at the bottom of the page. The user specifies the type of aggregate to be calculated (in this case a summation on the budget column) and when to calculate the aggregate. The aggregate can be calculated for every break column, for the whole report, or for each page (see Fig. 2-11).

When an aggregate is calculated, it is calculated for every break column if that option is selected. If both project and department are break columns, and the user wishes to calculate total budget only by project, RBF will not provide that capability. This is an instance where the Report Writer would be a more appropriate tool. Break columns are thus considered in more detail in the section on the Report Writer.

Another feature of RBF allows the user to have data selected at the time the report is run. Using this feature, the user can select certain columns in the report and specify that this column is part of the run-time data selection. When the report is run, the user is prompted for the value of that column. For example, a personnel report could be run selectively on certain departments or employees. The user can still have the report run on all departments by typing in a "*" for the value of that field.



```
RBF - Column Options

 Column Name:  budget                                   Break Column:  n



 Selection Criteria at run time:  n     (n = none, v = value, r = range)

          Enter "x" to select Aggregate/Break combinations for column.
               ┌─────────────────┬────────────┬────────────┬───────────┐
               │Aggregate        │Over Report │Over Breaks │Over Pages │
               ├─────────────────┼────────────┼────────────┼───────────┤
               │Any              │            │            │           │
               │Average          │            │            │           │
               │Count            │            │            │           │
               │Maximum          │            │            │           │
               │Minimum          │            │            │           │
               │Sum              │            │            │x          │
               │                 │            │            │           │
               └─────────────────┴────────────┴────────────┴───────────┘
  Help(PF2)  End(PF3)
```

Courtesy of Relational Technology

Fig. 2-11 Defining Aggregates for a Report

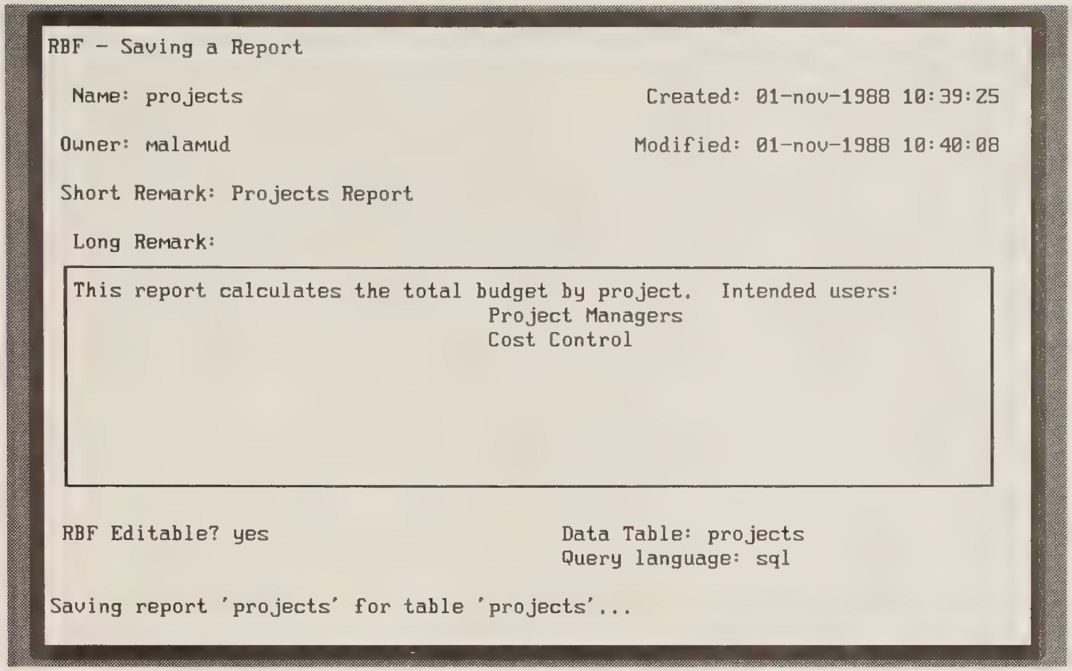
The last set of features the user can change in RBF is the overall output options. These specify the length of the page and margins. When a report is run, INGRES looks to see if the report is being displayed on a terminal screen or into a file or a printer. If the report is on a screen, INGRES sets the page length by default to 23 lines in a file or the printer to 61 lines.

When the user saves the report, the report specifications are stored in the database and form part of the catalogs for the report execution subsystem (see Fig. 2-12). The report can be annotated with long and short remarks. This information is added to the catalogs, such as information on the report creator, when the report was modified.

Other users can then pick the report from the report catalogs, just like they would pick a query target from the QBF catalogs. Users are then prompted for any run-time selection values, and the report is run (see Fig. 2-13).

VIGRAPH

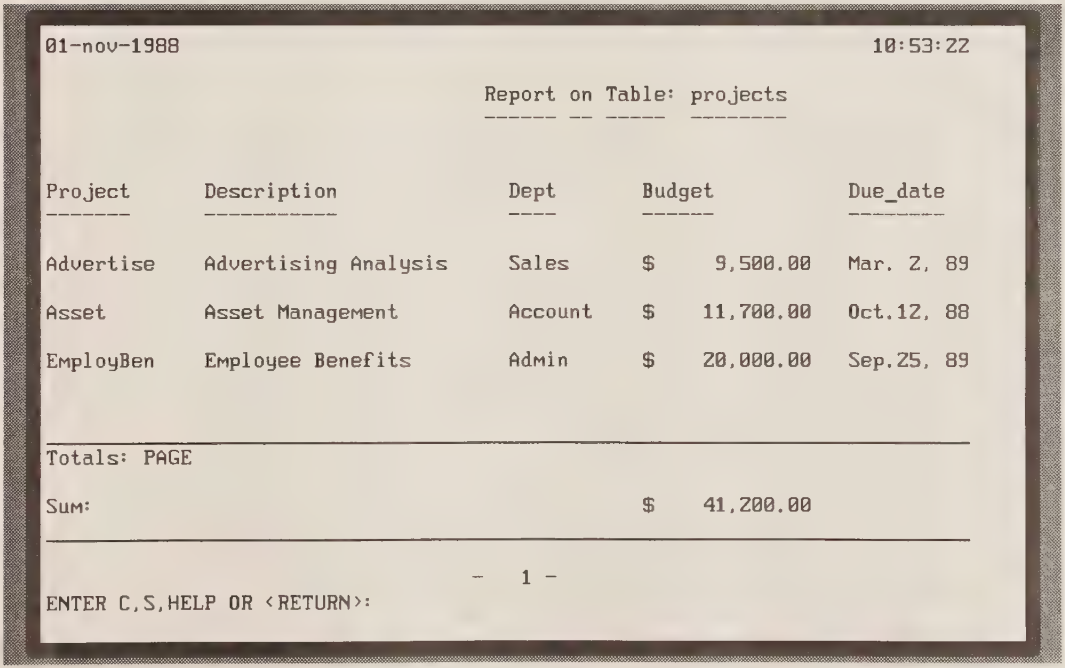
VIGRAPH is the third general method of examining data. QBF allows interactive browsing of data, and RBF allows the user to define reports on data. VIGRAPH is used to define graphs that display data in the database. Like RBF, VIGRAPH consists of two



```
RBF - Saving a Report

 Name: projects                                      Created: 01-nov-1988 10:39:25

Owner: malamud                                      Modified: 01-nov-1988 10:40:08

Short Remark: Projects Report

 Long Remark:
+------------------------------------------------------------------------------+
|This report calculates the total budget by project.  Intended users:          |
|                                  Project Managers                            |
|                                  Cost Control                                |
|                                                                              |
|                                                                              |
|                                                                              |
+------------------------------------------------------------------------------+

RBF Editable? yes                         Data Table: projects
                                          Query language: sql

Saving report 'projects' for table 'projects'...
```

Courtesy of Relational Technology

Fig. 2-12 Saving a Report in the Catalogs



```
01-nov-1988                                                               10:53:22

                                  Report on Table: projects
                                  ------ -- -----  --------


Project      Description               Dept       Budget           Due_date
-------      -----------               ----       ------           --------

Advertise    Advertising Analysis      Sales      $      9,500.00  Mar. 2, 89

Asset        Asset Management          Account    $     11,700.00  Oct.12, 88

EmployBen    Employee Benefits         Admin      $     20,000.00  Sep.25, 89


-----------------------------------------------------------------------------
Totals: PAGE

Sum:                                              $     41,200.00
-----------------------------------------------------------------------------

                                     - 1 -
ENTER C,S,HELP OR <RETURN>:
```

Courtesy of Relational Technology

Fig. 2-13 Running the Report

steps. First, the user defines a graph; then the user (or another user) points to a particular graph in the catalogs and runs it.

Unlike RBF and QBF, VIGRAPH requires a terminal capable of supporting graphics. While the other forms subsystems run on very dumb terminals (i.e., old and/or cheap), VIGRAPH requires a terminal capable of producing the basic components of a graph—circles and lines. It is possible to run VIGRAPH on dumb terminals, but letters, numbers, and punctuation symbols will be used to represent graphic objects, not a satisfactory solution. Note that this limitation is not especially severe—several dozen types of devices are supported instead of the many dozens supported for the basic forms system. It is also increasingly rare for users not to have access to either an intelligent terminal or a workstation.

A graph, like a form or a JoinDef, is an object that is stored in the INGRES system catalogs. The graph itself is made up of several different objects. These can be display objects, such as pies in a pie chart, lines in a line graph, or bars in a histogram. VIGRAPH also supports several other types of objects. The graph can have a legend, a title for the graph, or trim (explanatory text). Finally, fields can be displayed that are used to show information that is not charted. In a sales graph, for example, the name of the salesperson could be displayed in such a field, along with a bar chart comparing sales over time.

The graph itself has a variety of attributes that include the color of the background for the graph and the drawing color used for either the overall graph or subobjects such as pie slices in a pie chart. In addition to color, VIGRAPH supports hatch patterns, which are patterns used to fill subobjects. Hatch patterns are important when data are output to a noncolor device to enable the user to distinguish different subobjects in a graph.

Finally, the user can define fonts and line styles for a graph. A font is the type of text used to display information. Line styles allow lines to be specified as stars, dots, triangles, squares, or other objects. These components are marks for data values in a line chart. The user can also choose from a variety of connection types to connect different points on a line.

Since drawing a graph can be time consuming on many terminals, VIGRAPH allows different options, known as presentation levels, that control how much of a graph is redrawn whenever objects are edited. At the highest presentation level, all color and fonts are displayed. At the lowest presentation level, fonts are displayed as a box, hatch patterns are removed, and other efficiency mechanisms are used to quickly redisplay the edited graph.

Creating a graph begins by mapping a graph to a table or view in the database. This is similar to creating a default report in RBF from a table in the database, or using QBF on an object in the database. Mapping includes specifying the horizontal and vertical axes for a graph. The user positions the cursor on one of the fields and picks the *ListChoices* option to show the available columns.

Note that it is possible to do the mapping at a later point. VIGRAPH will then use default data until real data are mapped. It is also possible to change the data mapping on a graph. A user can take an existing graph in the catalogs, change the mapping, and save the graph with a new name.

Each of the two axes represents a series of data. To plot the budget for each project, the project would be the X axis and the budget series would be the Y axis (see Fig. 2-14). In addition to the two axes for the graph, the user can pick additional series of data. In a line chart, for example, the user may wish to plot several different series of information for comparison.

Once the basic data have been defined, the user begins to customize the graph (although he can stop at this point and run the graph as is). A graph can be resized and objects edited. For a bar chart, for example, the user can change the size of the bar chart or add explanatory text. Figure 2-15 shows the screen display for editing a graph on a VT100 terminal.

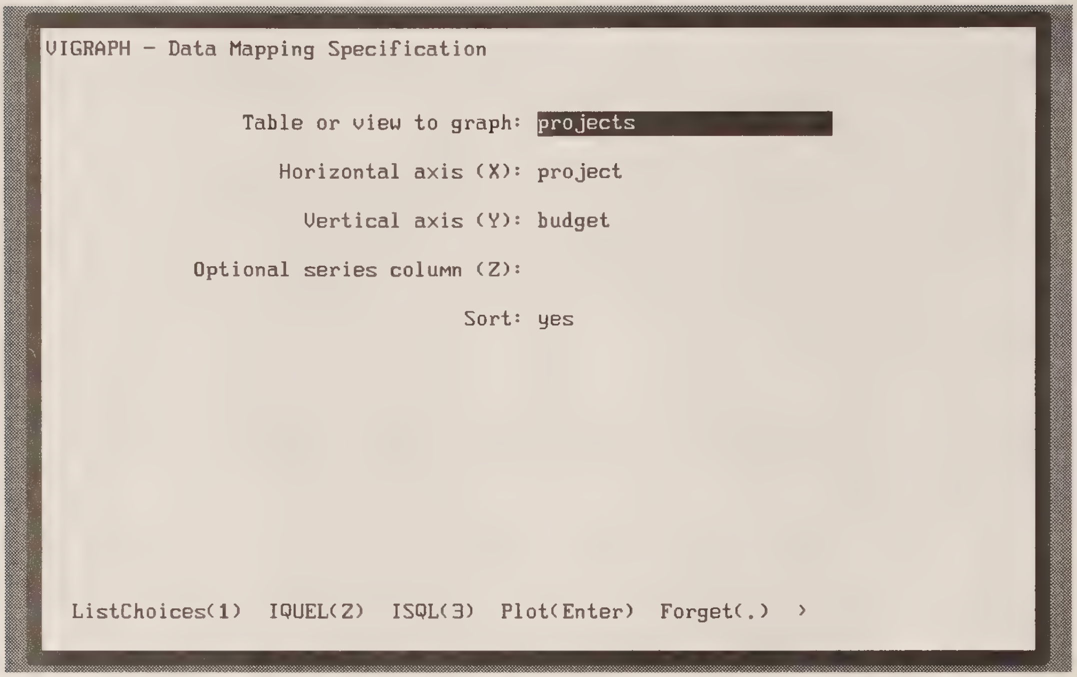
For pie charts, the user can choose to explode the pie. This involves picking one of the subobjects (a slice) and moving it out from the center. For a pie chart, the X and Y axes are both displayed within the pie. Normally the X value is a text label and the Y value is a percentage of the total pie. The user can edit the format of these labels.

Each of the objects on the graph has a series of attributes assigned to it. In the case of the bar chart, for example, the graph as a whole has attributes, such as background color (see Fig. 2-16). Each of the subobjects of the bar chart, such as each axis, also has attributes including the font and color for the text and the placement of the axis. At any time during the process, the user can plot the graph. This plot can go directly to the terminal or to some other device on the network. Figure 2-17 shows a completed plot of the bar chart on a PostScript display device.

VIGRAPH provides a great deal of flexibility for defining different forms of graphics display. In this sense, it requires a higher degree of skill to operate than RBF or QBF. Once the graph is defined, however, it stays in the database and any user is able to run it. The current version of the data are retrieved and displayed according to the format defined for that graph.

In many organizations, one or two users will be assigned to use VIGRAPH, and they might then define a standard set of graphs. Any other user can simply point to that object and have it displayed. The users of the graph don't need any knowledge of VIGRAPH, only the ability to point to an object and watch it being displayed.

It is important to note that no one in these classes of users—graph editors and graph viewers—needs to be a programmer. One of the advantages of this type of data access environment is that programmers or the MIS department do not need to be involved. Users can quickly formulate new graphs (or reports or JoinDefs) without the need for assistance from programmers, allowing the programmers to concentrate on their applications development. These three tools, graphs, reports, and QBF queries, are available to end users.

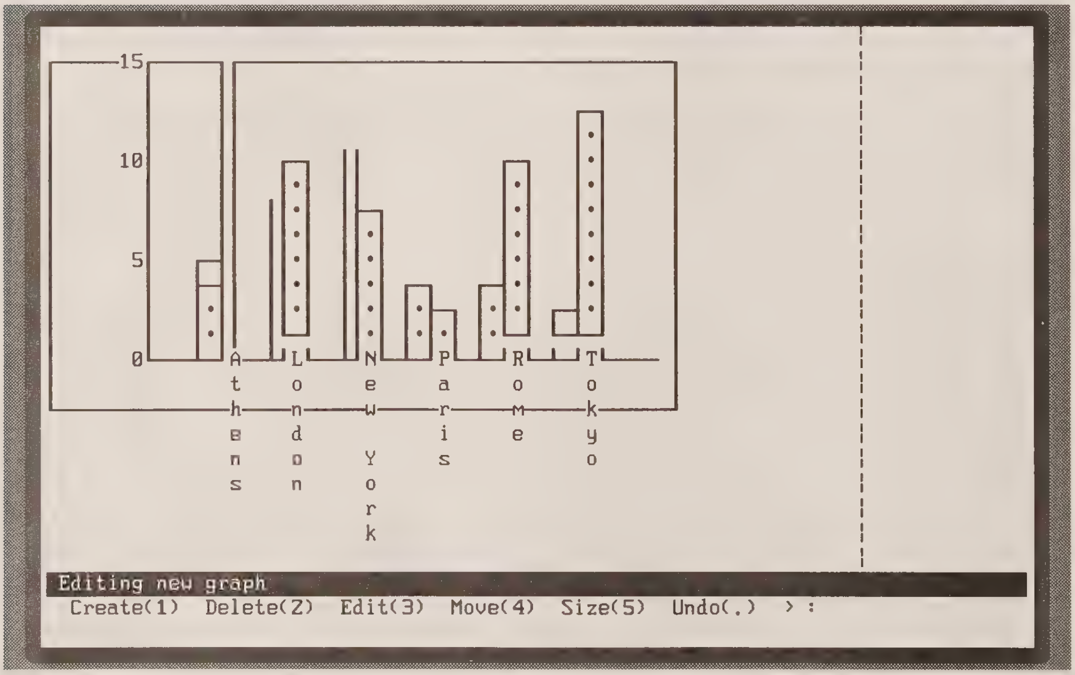


```
VIGRAPH - Data Mapping Specification

               Table or view to graph: projects

                  Horizontal axis (X): project

                    Vertical axis (Y): budget

         Optional series column (Z):

                                 Sort: yes
```

ListChoices(1) IQUEL(2) ISQL(3) Plot(Enter) Forget(.) >

Courtesy of Relational Technology

Fig. 2-14 Mapping the Bar Chart to a Table

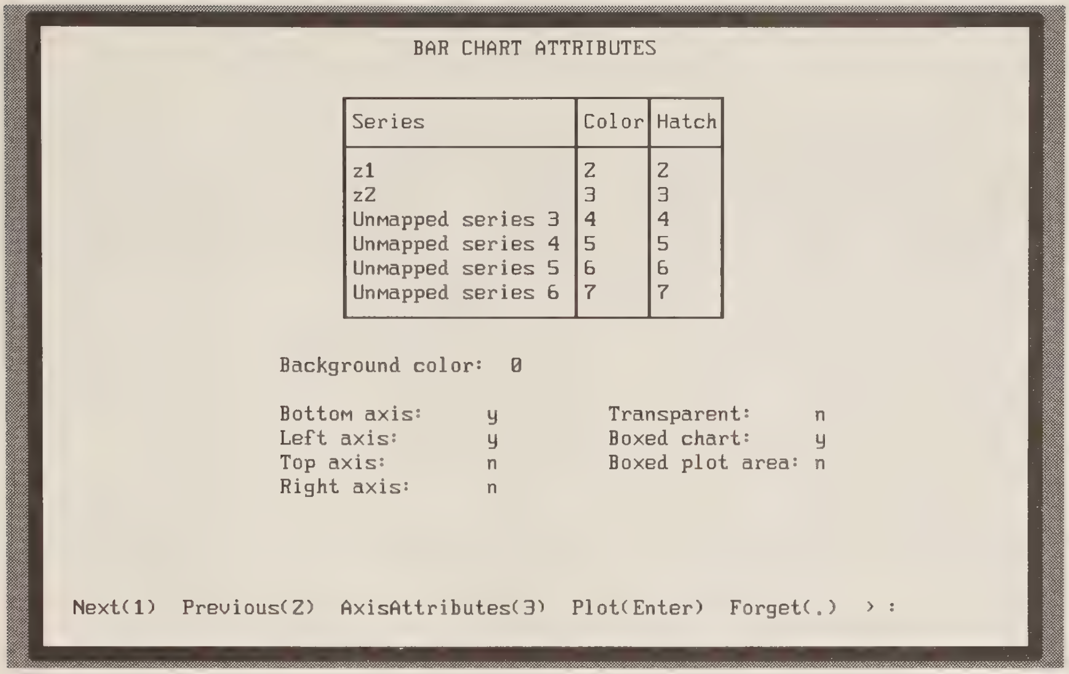


15
10
5
0
Athens
London
New York
Paris
Rome
Tokyo
Create(1) Delete(2) Edit(3) Move(4) Size(5) Undo(.) > :

Editing new graph

Courtesy of Relational Technology

Fig. 2-15 Editing a Bar Chart on a VT100 Terminal



BAR CHART ATTRIBUTES

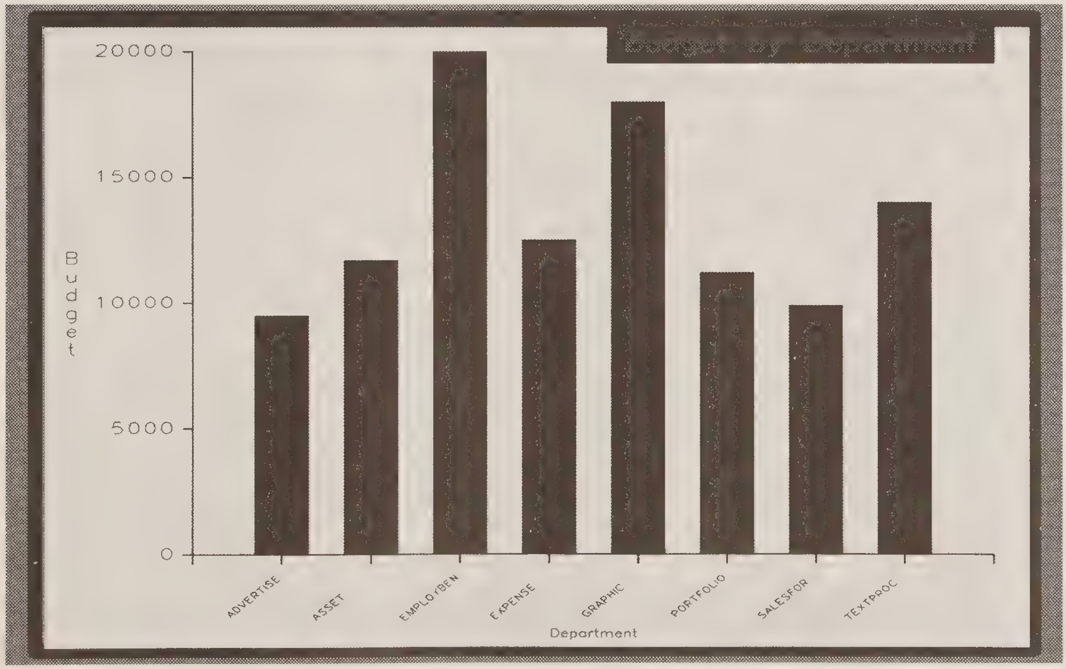
Series	Color	Hatch
z1	2	2
z2	3	3
Unmapped series 3	4	4
Unmapped series 4	5	5
Unmapped series 5	6	6
Unmapped series 6	7	7

```
Background color:  0

Bottom axis:      y         Transparent:     n
Left axis:        y         Boxed chart:     y
Top axis:         n         Boxed plot area: n
Right axis:       n
```

Next(1) Previous(2) AxisAttributes(3) Plot(Enter) Forget(.) > :

Courtesy of Relational Technology

Fig. 2-16 Attributes of a Bar Chart



Courtesy of Relational Technology

Fig. 2-17 Testing the Bar Chart with Data

Visual-Forms-Editor

When a user executes QBF on a particular JoinDef or table, QBF builds a default form for that query target. As previously shown, the user can choose simple fields or table fields. More sophisticated forms capabilities are not available in this default mode. VIFRED is used to prepare a form with more sophisticated visual and hidden attributes. Often VIFRED forms are used within the context of QBF. However, as will be seen, VIFRED is a more general tool than just a pretty form of QBF.

The form is the fundamental concept used in all of the INGRES subsystems, not just data entry and retrieval in QBF. When a programmer develops a new utility such as RBF, that utility consists of a series of forms. Each form has a menu associated with it that performs certain actions. The application development environment uses the facilities of VIFRED to generate the forms used for these new tools.

With VIFRED, a user can request that a default form be built from a database object, such as a table, view, or JoinDef. This default form is the same one the user would have seen if she executed QBF on that object. The user can then modify the form and store it in the database and the form then becomes part of the forms catalogs (see Fig. 2-18).

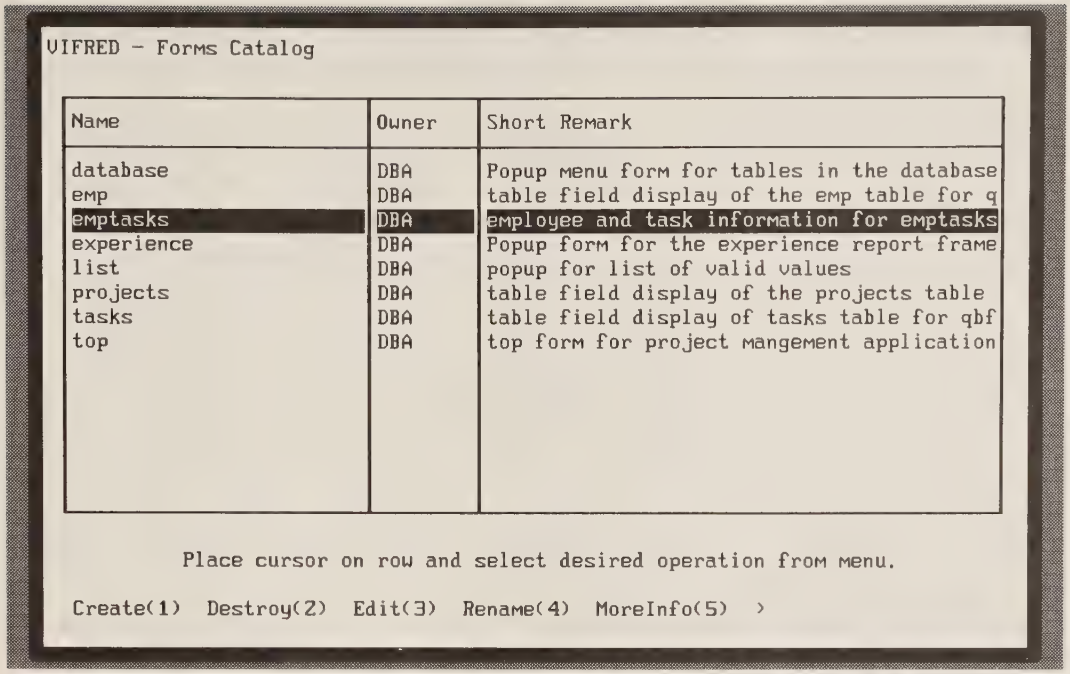
The form then becomes a new object. In a QBF environment, the user will want to use that form in conjunction with a table, view, or JoinDef. It is possible to use a single form on different tables if the tables have fields in common. It is also possible to use the same form on different JoinDefs—each one having different update or deletion rules.

Because a form by itself is not tied to a JoinDef or table, the user needs to make that association. The association between a JoinDef or table and a form is known as a QBFname. When the user starts QBF, he can choose three different types of catalogs: QBFnames, JoinDefs, and tables. The JoinDef and tables catalogs build a default form, while the QBFname catalog uses a specific form.

Once a form is displayed on the screen for editing, the user can change its appearance (see Fig. 2-19). Usually this consists of moving and altering the appearance of fields and trim. Trim is text not associated with a field. For example, the default form includes the table name as trim. Often, users will wish to replace the table name with a more expressive description.

Fields have both a title (a form of trim) and a display format. By default, the title for a field is the name of the column in the database, with the first letter capitalized. Last name might appear on the form as “Lname.” The user could change the title of the field to read *Last Name:*. The display format for a field governs the length and the appearance of data within the field. It is possible to make fields longer or shorter than the actual data in the database. If the field is longer, data are truncated when added to the database. If the field is shorter, data from the database are truncated. Remember, a form is not tied to the underlying table until QBF is actually run.

For long text fields, it is possible to specify a special format that specifies how many lines in the field and how words are to be treated at the end of a line. For example, if a



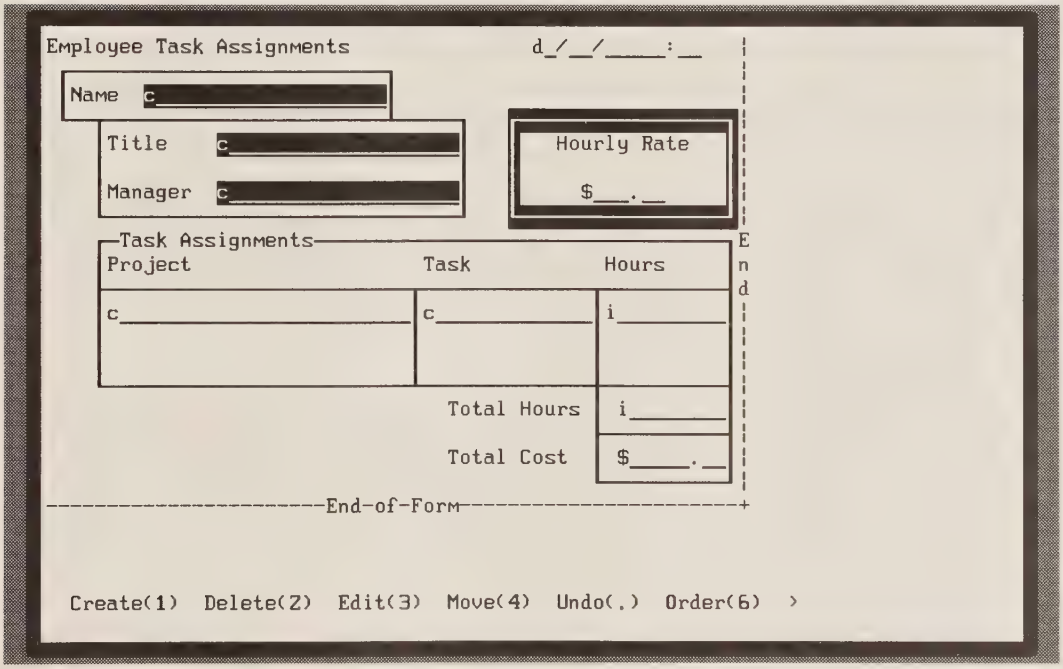
VIFRED - Forms Catalog

Name	Owner	Short Remark
database	DBA	Popup menu form for tables in the database
emp	DBA	table field display of the emp table for q
emptasks	DBA	employee and task information for emptasks
experience	DBA	Popup form for the experience report frame
list	DBA	popup for list of valid values
projects	DBA	table field display of the projects table
tasks	DBA	table field display of tasks table for qbf
top	DBA	top form for project mangement application

Place cursor on row and select desired operation from menu.

Courtesy of Relational Technology

Fig. 2-18 VIFRED Catalogs



```
Employee Task Assignments                 d_/__/____:__
Name c
Title c
Manager c
Hourly Rate
$___.__
Task Assignments
Project                  Task           Hours
c                        c              i
                    Total Hours         i
                    Total Cost          $_____.__
----------------------End-of-Form----------------------+
```

Courtesy of Relational Technology

Fig. 2-19 Editing a VIFRED Form Layout

column in the database is defined as a text field with 2000 characters, the default format for the field would be *c2000*, a single 2000-column simple field. The user could specify the format for the field to be *cj2000.50*. This format specifies display of the field in lines of 50 characters and right justification of the text when it is displayed.

In addition to the data format and the title, a field has a series of attributes. Visual attributes govern the appearance of the field, including reverse video, blinking, underline, or color. A special type of visual attribute is the no echo characteristic, which says that data typed in the field are not visible on the screen. This is used, for example, to enter passwords.

Other attributes govern how a field can be used. The mandatory field attribute specifies that data must be entered into this field. This is useful where the field uniquely identifies data in a row, social security number in a personnel application, for example.

Two attributes help the user in data entry applications. The default value attribute automatically fills in a value on the screen. For example, an insurance company in California would usually issue policies within that state. The default value for the field "STATE" could be California. The second attribute allows the previous value in a field to be retained instead of being cleared after each append operation. This is useful when batches of data from the same source are being entered.

The display only attribute allows a user to view data but not change information in the field. When the user on a previous field hits the tab key, she skips over this field and goes to the next one. The query only attribute can be used to allow a field to be used for queries, but otherwise has the display only characteristic.

Two attributes of a field help ensure that data are entered properly. First, the field can have force upper- or lowercase enabled on it. Any data entered are automatically converted to the appropriate case. This prevents the problem of state names being entered in lowercase and then not being retrieved in a query with uppercase.

The second, more complex, attribute is the validation check. The validation check is performed whenever the user tabs off of the field. There is a message associated with each validation check. Whenever the validation check fails, the message is displayed on the bottom of the screen. A validation check has two possible forms:

- simple comparisons
- set comparisons

A simple comparison checks the value of a field against a constant. These simple comparisons can be used with boolean operators to check multiple values:

```
lname = 'M*' or lname = 'm*' and not lname = 'Bozo'
```

Comparison checks can also use bracketed values:

```
lname = '[ABCDE]*'
```

The more complex form of validation checks compares a field against a set of values, either specified by the user or contained in a database table. For example, if the name of the field is state, the user can define a database table called valid_state with a column called abbrev. Then, the following validation check can be defined:

```
state in valid_state.abbrev
```

This is an extremely powerful mechanism for ensuring that data entered meet the specified set of valid entries. In fact, when this type of validation check is defined and the user hits the HELP key in any of the INGRES subsystems, the list of valid values is displayed. Keeping valid values in a database table allows new criteria to be established without redesigning forms and other parts of applications. A data entry application can be kept flexible to changing circumstances.

One potential disadvantage to using this form of validation check is that the valid values are retrieved from the database when the form is first displayed to the user. This has two potential drawbacks. In a transactions processing environment, the list of valid values could change rapidly. The user would not see the values that had changed since the form was displayed. If this were the case, the applications developer would perform the same type of validation check using the INGRES 4GL instead of building the check into the form. This would allow the values to be compared at run time.

The second potential drawback of reading the values into memory is that validation checks on very large tables become unwieldy. If there are 10,000 possible state abbreviations, all 10,000 valid values would be read into memory. On a large form with many fields of this sort, the amount of memory used would be prohibitive. Again, the solution to this problem would be to build the validation check into the INGRES 4GL code instead of into the form.

Whenever a validation check is built into the INGRES 4GL code, this means that it is not available from QBF. Instead, the VIFRED form would be used in the ABF development environment. This is an example of a custom application necessary in a specific instance. Again, if QBF is suitable, it is not necessary to move toward the next level of a custom application. Instead, a VIFRED form and a QBF JoinDef could be used.

As will be seen in two chapters, it is possible to combine canned interfaces (i.e., QBF and VIFRED forms) with custom code into a single application. This means that the application developer need only write the code for those particular parts of the application that need it. For the rest, the developer simply specifies a particular QBF operation.

Report Writer

The Report Writer is used to generate sophisticated reports that are beyond the capabilities of the RBF subsystem. In the Report Writer, the user edits a file with a series of commands in it, and then executes the *sreport* command, which reads the text file with the commands and builds a report specification that is loaded into the database. The file is loaded into the database using the sreport command. Once the report is part of the database, it can then be run using the report catalog along with RBF-developed reports.

A report in the Report Writer consists of several different sections, each of which has a series of commands. The first section specifies what data will be used for the report. This consists of a query in SQL or QUEL, or the name of a table or view. This is the first instance where the Report Writer is more general than RBF. Because any arbitrary query can be specified, it is possible to have highly complex data selection.

It is also possible to parameterize a query. A parameter is read in when the report is actually run, and the value of a parameter is substituted into the query. Usually, parameters are used for values in the where clause of a query:

```
select * from emp where emp.name = $name
```

When the report is run, the user is prompted for the value of name. Sometimes, the parameters are filled in on a form. The form, developed in VIFRED, can have validation checks and default values to make sure that the appropriate parameters are substituted. This topic is discussed in more detail in the next chapter on ABF.

The other sections of a report correspond to break columns. A break occurs whenever a value changes. Data in a report are sorted by columns. In a personnel report, we might sort data first by department, then by employee name, finally by project. For each project there could be several rows of data.

Whenever a new department value occurs, this is known as a break on the department column. A common break action for this report would be to go to a new page for each department. The Report Writer allows a series of actions to occur either before or after a break occurs. Going to a new page for a new department is an example of a header action for the department column.

A footer action occurs when the last row for a particular break column occurs. A break action for the department column might be to print the total value of salaries for the department. To specify this action, the following code would be placed in the report file:

```
.footer department

        .newline
        .print sum(salary)
```

Notice that the sum for salary is calculated over that department, as opposed to the whole report. All aggregates are always calculated within the context of that particular break column.

Two implicit breaks are for pages and for the entire report. Whenever a new page occurs, either because the page is full or because there is a .newpage command in the code, this generates a break on page. The report code can have a .header or .footer section for the page. In a header section, the user might specify column headings, the date the report was run, or other information. In the footer section, it is common to put the page number of the report.

The last major section of the report is the detail section. The detail section is executed for every row of data. A report might have the following sections:

```
.query
        select * from emp
        .sort department, employee, project

.header report

.header page
        .print currentdate
        .newline
.header department
        .newpage
        .print department
.header employee
        .print employee
.header project
        .print project
.detail
        .print tasks, hours

.footer page
```

This report is missing several important formatting commands that position the data in the appropriate column. However, the major portions of the report are in place. By placing the .print department command in the header for department, that piece of data is only printed once. If the department column were printed in the detail section, it would be repeated once for each task.

Break columns are thus used to print data only where appropriate, as well as to calculate aggregates within the context of a break column. The detail section is used to print the basic data one row at a time. Note that each instance of the detail section of a report could be several lines (or pages) of the report.

Within any of these sections—headers, footers, or the detail section—the user can specify a wide variety of different commands that specify exactly how data are to be printed. The user typically moves to a particular position on the page and asks for data to be displayed. Printed data can be a value from the query, text strings, or functions in the Report Writer.

Data can be printed using a default format, or using any the templates that are used in RBF. Text value is simply printed by enclosing it in quotes. Functions can print the current date, the page of the report, or a variety of other information.

A typical header for a page might consist of the following:

```
.header page

        .newline 3
        .tab right .print current_date
        .newline
        .print "Report on Employee" .tab+3 .print employee
```

Positioning of data columns and the format for those columns can also be included in a special setup section of the report. The user specifies the format, consisting of a template or format (i.e., c20), and the position on the page for that column. Then, the user can simply execute the following:

```
.tab employee .print employee
```

Other setup commands include a short remark and a long remark for the report. These items are displayed in the report catalogs so that the user running the report knows the purpose of the report. Finally, the user can specify the default output device (i.e., a particular printer or file name) for the report.

Some more sophisticated capabilities of the Report Writer include:

- arithmetic operations and internal variables
- if statements
- column and block commands

Unlike RBF, the user can perform arithmetic operations on data within the Report Writer, instead of defining a view to perform these operations. Rather than define a view with total price equal to unit price times quantity, the same operation can be performed in the Report Writer:

```
.print unit_price*total_price
```

Internal variables are used to keep track of information that does not come from the database. Counters are an example of this. A page number is a default variable in the Report Writer. On the other hand, the user may wish to keep track of the number of big sales and print that information at the bottom of the page. The report would include a definition for this counter in a special declare section of the report. Then, the values of the counter would be incremented every time a certain value of sales is exceeded. Finally, the value would be printed at the bottom of the page:

```
.declare
        counter = integer

        .detail
        .if sales > 10000 .then counter = counter+1
        .endif
.footer employee
        .if counter > 0 .then
                  .print counter," sales greater than 10,000!!"
        .else
                  .print "Poor performer award."
        .endif
        counter = 0
```

This example also illustrates the use of the if statement to test values in the report and to change how the report executes depending on the values. This is the type of operation that needs the capabilities of the Report Writer instead of RBF.

Internal variables, like information in the query, can be read in at run time. This allows parameterized variables to be used in if statements. In our previous example, the target salary can be read in when the report is run instead of hardcoding that information into the report specifications:

```
.declare
        counter = integer,
        target_sales = money with prompt "Target sales?"
.detail
        .if sales > target_sales .then counter = counter + 1
```

The most sophisticated (and complex) capability of the Report Writer is the ability to define subregions of a report—blocks and columns. The need for this is best illustrated in the example of converting an existing information system to an INGRES environment.

Developers learn from hard experience that the first thing that users want to see out of a new information system is an exact replica of all existing reports. No matter that INGRES can produce a more desirable format than the existing system—the initial test is the ability to exactly duplicate the existing report.

In the Report Writer, information is printed on one line, then the next, and so on down the page. In the department report example, the header is first executed, then the detail, then the footer section. The footer section would go on the bottom of the page. What if the user wants to see the total salary for a department at the top of the page? This is a footer action.

Blocks and columns allow virtual areas to be defined on the page. The Report Writer can then move up and down these areas. This allows the footer action for department to move up the page, print the aggregates, then move back down to the previous area.

The Report Writer is a language-based environment, in contrast to forms-oriented systems such as RBF. As in the case of a C program, the report developer uses a text editor to prepare the file and then loads the report into the database using the sreport command. Because the Report Writer requires the user to learn the syntax of the language, it requires a higher level of sophistication than RBF. Unlike QBF, RBF, and VIGRAPH, this subsystem requires a little more training. Just because the Report Writer is at the next level of sophistication doesn't mean that the tool should be locked up in the MIS department. Most end-user departments have users with various levels of sophistication. This is a tool appropriate for the more sophisticated end users.

Terminal Monitors

The last subsystem that users may wish to use are the terminal monitors. Terminal monitors differ from the other subsystems, particularly QBF, in that the user directly enters SQL (or QUEL) statements and sends them to the data manager. The role of QBF was to present a more intuitive user interface and to generate the SQL on behalf of the user.

Directly entering SQL statements is often performed by two classes of users. First, programmers often use a terminal monitor to test SQL statements before embedding them into the 4GL or a conventional programming language. It is also possible to run the terminal monitor with a form of diagnostics that shows exactly how the query is executed, known as a query execution plan. Programmers would use this facility to try and improve the performance of queries.

The second class of user that would use a terminal monitor is the database administrator (DBA). The DBA uses the terminal monitor to set up security and integrity on the database, facilities that are not available from QBF. Like QBF and RBF, the terminal monitor is a forms-oriented application. However, it is a fairly simple one. Rather than interpret the user input, it sends it off directly to the back end, and, rather than provide processing of the data coming back, the terminal monitor simply shows it to the user.

Summary

The subsystems and operations discussed in this chapter are often all that INGRES users will see. QBF, RBF, and VIGRAPH allow fairly sophisticated operations on data to be performed. For most applications, this is enough.

QBF allows the user to append, retrieve, and update data in the database. A default form is built on the target of the query. Users can then choose various selection criteria to specify exactly what rows they are interested in viewing.

The JoinDef is an object used in conjunction with QBF to define how multiple tables are combined in a single operation. The JoinDef specifies which tables are master tables and which ones are details. In addition, the JoinDef defines the rules for update and delete operations.

Both tables and JoinDefs are objects that are used in conjunction with QBF. When a VIFRED form is combined with a table or JoinDef, this is another type of object used with QBF—the QBFname. All three targets, or objects, are stored in catalogs. The user simply points to the object in the catalog and specifies the type of operation to be performed—retrieve, append, or update.

RBF allows reports to be defined. Again, the report is an object stored in a catalog. The user points to the appropriate report and asks for it to be run. When running a report, the user can specify if output is to be put into a file, displayed on the screen, or printed. The user is also prompted for any parameters, that is, values read in at run time to narrow the selection criteria for the report.

VIGRAPH is the third subsystem used to examine data in the database. Unlike the other two, it allows data to be displayed in a graphical form. Once again, a graph is defined and stored in the database as an object. The user of the graph utility is shown the objects in the catalogs and points to one to be run.

Some users will move to the next level of sophistication, using VIFRED or the Report Writer to define more complex operations on the data in the database. Using VIFRED, validation checks and other attributes of fields can be defined. Using the Report Writer, complex formatting of data retrievals can be defined.

It is important to note that none of these tools require extensive programming skills or application development. The tools are ready to go and the user simply picks the appropriate query target and begins work. These tools thus allow users direct access to their data and are ideal for ad hoc analysis.

Chapter

3

Application Development Environment

Applications and Objects

The previous chapter considered general-purpose user interfaces such as QBF. These applications provide users with a method of accessing and manipulating data in the database. Application-By-Forms (ABF) allows users to develop their own custom applications. Instead of seeing the QBF menu, for example, a user of the custom application would see a menu developed by a particular organization.

ABF is the environment that the application developer uses to define the various objects that make up the final application. ABF is thus a shell for the application developer, just as INGRES/MENU was a shell that allowed access to the general-purpose applications.

After defining the objects in an application, ABF constructs a program. The application developer installs this program in some central location, and the program is run by users as QBF or VIGRAPH would be run. Often the application is the only thing that users will run. It is possible to install the application so that it is automatically run whenever the user logs onto the system. In this way, the custom application replaces the operating system as the fundamental method of interaction with the computer.

ABF has three functions:

- a code management system
- a dynamic test environment
- a shell for access to other subsystems

As a code management system, INGRES spares the programmer from worrying about the location of different portions of the source code that eventually make up the completed program. The location of libraries, source files, object files, editors, and other subportions comprising the completed image are all transparently accessed by ABF. This allows the programmer to focus on the logical flow of control within the application and the testing of modules.

As a test environment, ABF is able to properly construct the completed program. This can be done dynamically, to test modules as they are being developed, or in a static manner for the final construction of the program. ABF again takes care of the proper execution of linking options and other complex aspects that make up the completed image.

As a shell, ABF allows programmers to access other subsystems used to construct portions of the applications. This includes access to an editor of the programmer's choice, as well as access to the Report Writer, VIFRED, QBF, and the VIGRAPH Graphics Editor.

The normal ABF procedure consists of the following steps:

- create an application
- define frames and procedures
- test frames
- create an executable image
- run the application

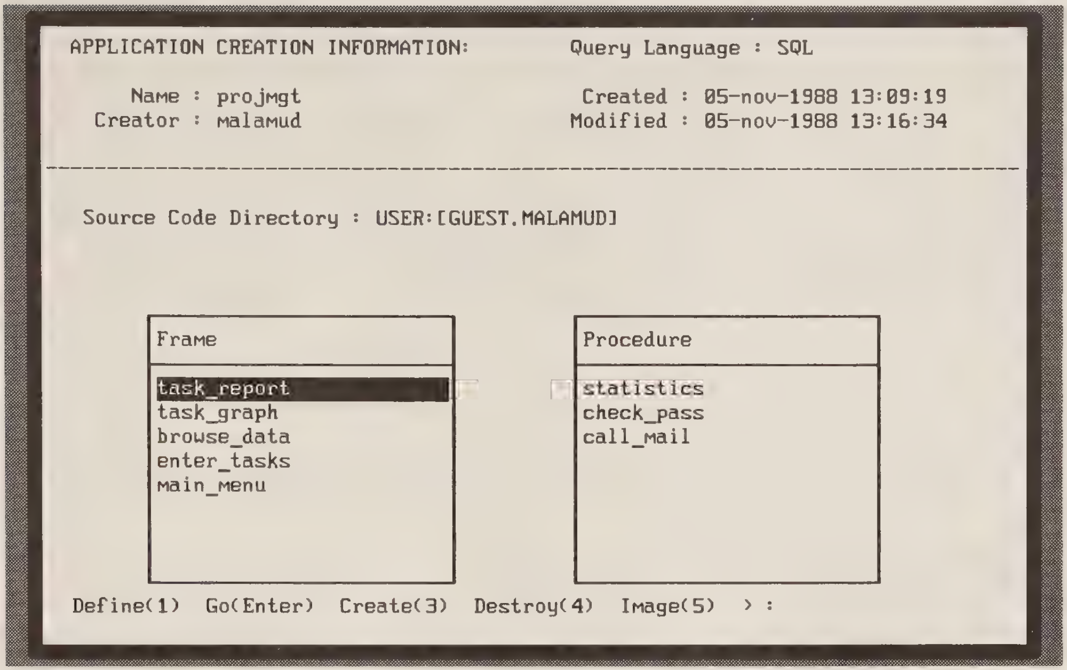
One advantage of an environment such as ABF is that it allows rapid prototyping of applications. Pieces of the application, such as menus and forms, can be put in place without having to define all of the details. Then, an executable image (program) can be created. The user can evaluate whether or not the flow of control in the application makes sense.

Once the overall design is approved, the developer can start filling in the pieces. If a menu lists seven kinds of reports, the developer might develop three or four of these reports. The application is again turned into a program, and users reevaluate the system. Finally, the remaining pieces are put into place.

This rapid prototyping environment has two important effects. First, development is quicker because the programmer is not forced to write all of the code—ABF generates a great deal of the code for the programmer and the application can run without all of the modules in place. Programmers begin focusing on the semantics or content of their application instead of writing large amounts of code to open files, refreshing screens, or coding other low-level functions.

Second, because development is rapid, it is possible to show users what the system looks like in the intermediate stages. This allows the design to be changed to reflect changing user requirements. Prototyping is in sharp contrast with more traditional development styles, where user requirements are fixed in the beginning stages and the programmers develop and test the final application before showing it to the users again.

Having users see intermediate results is important because requirements change continually in most environments. With ABF, even a completed application can be easily changed to reflect new information needs. In addition, most users are unable to look at a paper design document for an application and get a good picture of what the final



```
APPLICATION CREATION INFORMATION:        Query Language : SQL

     Name : projmgt                        Created : 05-nov-1988 13:09:19
  Creator : malamud                       Modified : 05-nov-1988 13:16:34
--------------------------------------------------------------------------------

  Source Code Directory : USER:[GUEST.MALAMUD]


          Frame                              Procedure

          task_report                        statistics
          task_graph                         check_pass
          browse_data                        call_mail
          enter_tasks
          main_menu

Define(1)  Go(Enter)  Create(3)  Destroy(4)  Image(5)  > :
```

Courtesy of Relational Technology

Fig. 3-1 ABF Main Panel

program will look like. Prototypes alleviate the need for users to form a mental picture of the application and allows them to see a concrete example of the system.

An ABF application consists of seven kinds of objects:

- application
- frame
- procedure
- form
- table
- report
- graph

Each of these objects has a series of attributes, such as fields on a form. These objects are kept in the INGRES system catalogs; by storing objects in the system catalogs, back-end services such as security, recovery, integrity, and controlled shared access are all shared.

The application is the overall object. It has a creation date, an owner, and several other attributes. The application also has objects that it owns, such as frames and procedures. Those objects, in turn, are made up of subobjects. Figure 3-1 shows the attributes and objects for an ABF application.

A frame is the basic INGRES construct. Each frame has a form and a menu associated with it. These frames are identical to the general-purpose user interfaces previously examined, such as QBF. In fact, it is fairly simple to reconstruct any of the general-purpose user interfaces in the ABF environment.

Menus are manifestations of actions that take place within the frame. The menu can consist of visible menu items seen by the user, or nonvisible activations such as a timeout, a field activation, an initialize block, or key activations. The code associated with a particular menu activation can include a further level of menus, that is, a submenu.

A QBF frame is an example of a general-purpose frame used in ABF. The frame consists of a form and an associated database object, such as a JoinDef or a table. The QBF frame has nothing but default operations, meaning that the programmer doesn't need to generate any INGRES 4GL code.

A report frame consists of a report, an optional menu, and a form used for any parameters passed into the report. The form, developed in VIFRED, allows the user to enter parameters. The fields on the form have the same internal names as the parameters in the report. This method allows the power of VIFRED validation checks to be used to control the entering of parameters for reports.

Graphs and procedures are two other forms of objects within an ABF application. A procedure can be written either in the INGRES 4GL or in a 3GL (possibly with embedded SQL statements).

Fourth-Generation Languages: INGRES 4GL

The INGRES fourth-generation language (INGRES 4GL) is the language used to define applications. Fourth-generation languages operate at a high level—a single command is able to accomplish a great deal. This is in contrast to more traditional third-generation languages that require the programmer to perform a great deal of low-level housekeeping such as opening files, declaring variables, and writing to the screen.

An example of a high-level operation in INGRES 4GL is the "HELP" command. When a user picks the HELP menu item, the application developer issues a *help_frs* command. Associated with this command is a file that contains help text for this particular operation.

When this command is executed, the user is given three options. First, the user can pick the WhatToDo option; when this option is chosen, the help text is displayed on the screen. A user can scroll up and down pages in the help file or can search for a particular string in the help file. Second, the user can see what different keys on the keyboard do. For example, the PF9 key might be mapped to the "SAVE" function—save the current work on the form to the database. The third option on the screen is help on the help system.

All of these different aspects of the help system are automatic—they required no programming by the application developer. A single command, calling the help subsystem, has a number of built-in functions. The programmer did not have to write the code for all these different aspects of the help command.

INGRES 4GL has three main types of functions. First, it provides a means of accessing the database in a large variety of ways. Second, it provides control over the Forms Run-Time System (FRS). Using FRS commands, the programmer can clear the screen, change the appearance of the form, or make the terminal beep.

Third, INGRES 4GL provides various types of control over how commands are executed. For example, when a menu item is picked, a certain block of INGRES 4GL code is activated. Within that block of code, the programmer can loop through blocks of code or do conditional execution based on the data entered. Another type of flow control is calling another object, for example, calling QBF when a RETRIEVE menu item is picked.

Activations

Every frame in ABF has a menu associated with it. The selections in the menu are known as activations. When the user picks a menu item, a block of code is activated and a series of operations is carried out. An example of an activation is the QBF retrieve operation. The user fills out the form and then hits the GO menu item.

The GO menu item goes to the form on the screen and takes the values entered into the fields to construct a query. That query is sent to the data manager. The first row of values is then put on the screen. It is possible for an activation to have no interaction with the form. An example of this is the QUIT menu item; when the user picks this, the application is exited.

In addition to menu items, there are several other kinds of activations. A field activation is a block of code that is run whenever the user leaves a particular field on the displayed form. A field activation might be used to supplement or enhance the VIFRED validation with a more complex validation.

Another use of the field activation is to fill out values on the form for the user. If a user tabs off of the employee field, the INGRES 4GL code can examine the value in the field and automatically fill out the address and phone number for that employee if it exists in the database.

A third type of activation is a key activation. The HELP function can usually be picked by either picking the menu item or hitting a special function key on the keyboard. The program does not refer directly to the keys on the keyboard. Instead, the programmer designates the key by a logical function called an FRS Key. This is then translated onto a specific key on the keyboard using a mapping file.

The first level is the logical level. The programmer designates a certain FRS key and ties it to a block of code (say the HELP function). Each terminal type then has a mapping file that maps the logical key to a physical key—the PF2 key on a VT100 keyboard, for example.

HELP is one example of an FRS key. Other FRS key logical functions are QUIT, SAVE, FIND, UNDO. There is also one FRS key for each of the menu items on the bottom of the screen. The programmer ties a block of code to these logical functions. Presumably the HELP function is tied to the code that performs the same function, although the malicious programmer could tie the HELP key to the QUIT code.

FRS keys are mapped to physical keys using mapping files. Mapping files permit a user to see the same key used for the same function across all of the INGRES subsystems. HELP is always the PF2 key for one user, always the CTRL/H key for another.

Several different mapping files can exist. Normally, there is a mapping file for each type of terminal. Many of these are supplied by Relational Technology; others can be developed by users. There is also a mapping file that applies to an entire installation. Finally, individual users or applications can have their own mapping files. The application developer is also able to temporarily change the mapping of certain keys within an application.

Mapping files have two important benefits. First, the user always sees the same physical key used for the same logical function. Second, the programmer is able to design an application without worrying about what kind of terminal the users will have.

There are thus three types of activations that a user will see:

- menu items
- key activations
- field activations

The final type of activation is a block of code that is executed every time a frame is called. This block of code is called the initialize block. An initialize block might be used to retrieve some data from the database and have the data appear in the form when it is first displayed. The application developer specifies these activations and their associated commands in a text file. This file is then associated with a user-defined frame that also has a VIFRED form associated with it.

If the only purpose of the frame is to present a menu and have the user pick an option, the VIFRED form may have no fields on it. The user need not fill in any options on the form, only pick the appropriate menu block. More sophisticated user frames are used to interact with a form that has fields as well as text.

A basic INGRES 4GL program might consist of the following:

```
'ADD_DATA' = {
                callframe enter_tasks ;
    }

'REPORT' = {
                callframe task_report ;
    }
'EXIT' = {
                exit ;
    }
```

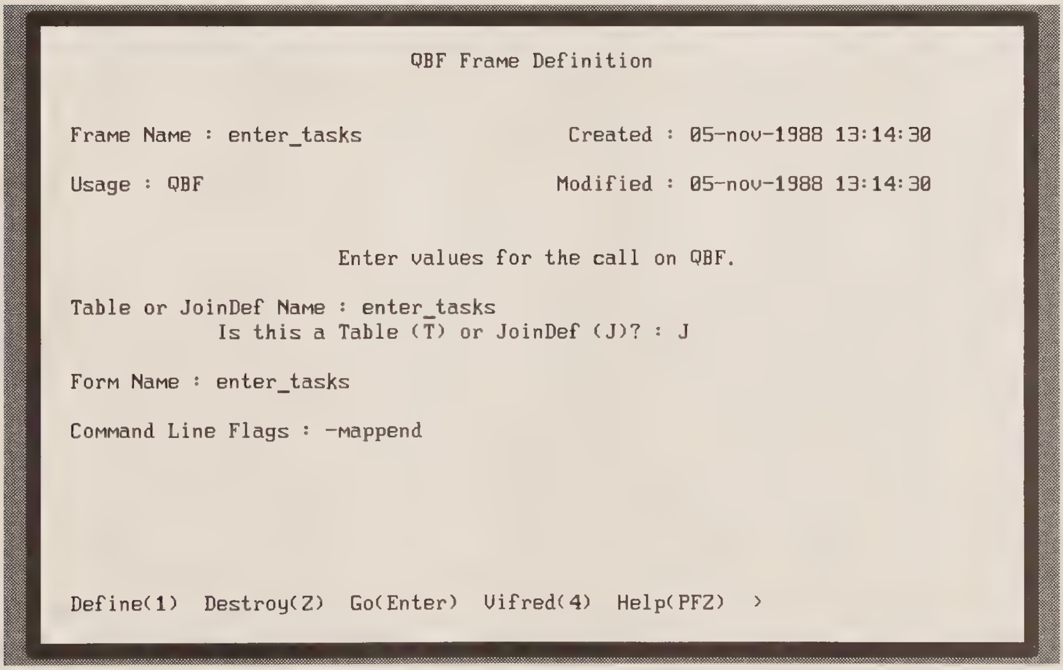
In this example, there are three menu activations. The first two activations will call the frames *enter_tasks* and *task_report*. The last menu item exits the application. Normally, the VIFRED form associated with this code would provide a little more explanation as to what these operations do.

Calls to Basic Subsystems

The callframe enter_tasks statement in the previous example calls a frame. This frame could be another user-defined frame. The developer in this case might define the frame enter_tasks to perform QBF-like functions, but customized to the particular needs of this user.

If QBF is an appropriate mechanism, there is no need to write a block of code emulating QBF. Instead of defining enter_tasks as a user frame, the developer can define a different type of frame, the QBF frame.

Rather than writing INGRES 4GL code, the application developer simply fills out the QBF frame definition. A query target is defined that can be a table (or view) or a JoinDef. A form is associated with the query target. Figure 3-2 shows an example of a QBF frame being defined. The developer specifies the enter_tasks join definition in the report and a form of the same name. In addition, the developer adds a command line flag that specifies that QBF should be called in append mode.



QBF Frame Definition

Frame Name : enter_tasks   Created : 05-nov-1988 13:14:30

Usage : QBF   Modified : 05-nov-1988 13:14:30

Enter values for the call on QBF.

Table or JoinDef Name : enter_tasks
Is this a Table (T) or JoinDef (J)? : J

Form Name : enter_tasks

Command Line Flags : -mappend

Courtesy of Relational Technology

Fig. 3-2 Defining a Call to QBF

The VIFRED form associated with the JoinDef has presumably been constructed with an append operation in mind. The form can have default values and validation checks to ensure that the proper data are entered in the database. The JoinDef controls which data will go into which database table.

When the user leaves QBF after performing a series of appends, the next screen displayed is the one that is called the QBF frame. Using the current example, the user would then see the main menu for the application. If the command lines had been left off, the user would have had a choice of the three different QBF operations: append, retrieve, and update. The developer thus chooses exactly which portions of QBF are to be incorporated into the custom application. There might be three different menu options, each of which calls QBF on the same query target using a different command line flag for each of the three operations.

While defining this QBF operation, the application developer also has access to two other important INGRES subsystems. First, the developer can call VIFRED. It is not necessary to define the form before defining the application. Often, the first thing defined is the INGRES 4GL code; then, VIFRED is called to construct an appropriate form for the INGRES 4GL code or the QBF query.

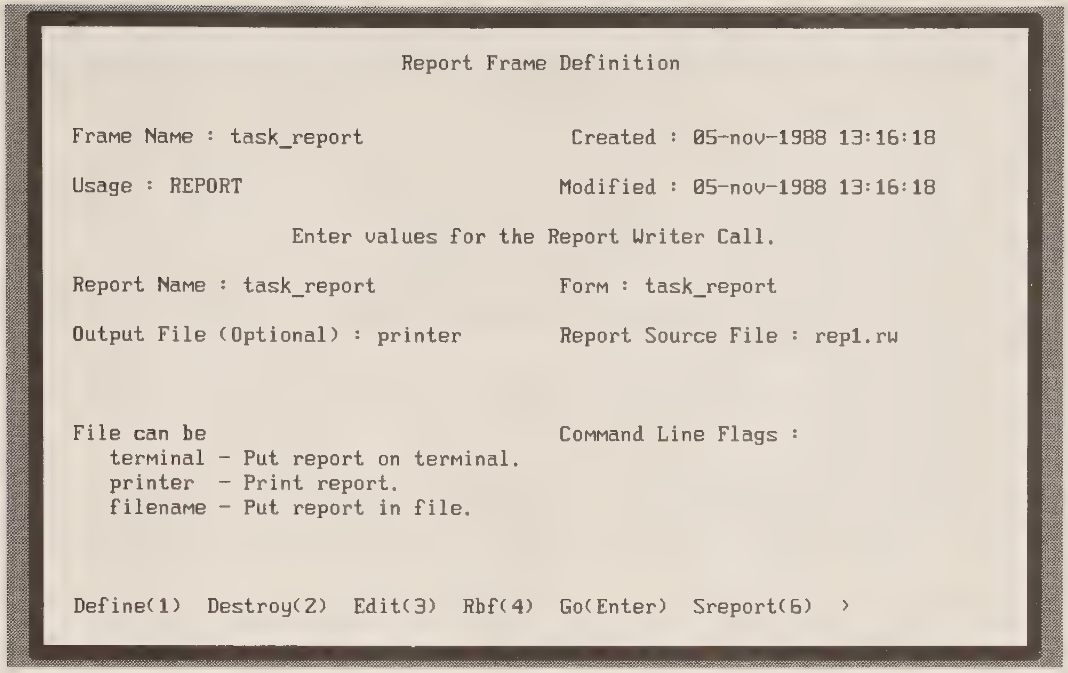
After the user is done defining the VIFRED form, a source code version of the form is moved into the applications directory. This source code version consists of either C or VAX MACRO assembly language code. When the application is compiled, this source code becomes part of the completed application.

It should be noted that the amount of source code generated when a form is compiled is considerable. The message to the programmer ought to be clear—it is much easier to use VIFRED to generate this code than to do it from scratch! Because this source code becomes part of the application, it will run significantly quicker than if the application had to go to the database system catalogs and dynamically construct the form. Note that if for some reason the form doesn't exist, then a default form will be constructed.

In addition to the VIFRED menu option, the QBF definition frame includes a GO menu option. This option allows the developer to immediately test a frame. A call will be issued to QBF using the specified form, query target, and command line options. Even though the rest of the application has not been tested, the developer is still able to make this portion execute as it would in a completed program. When testing is completed, the develop is returned to the QBF definition frame.

Another type of special-purpose frame is the report frame. As with a QBF frame, the developer calls this frame from a user-defined frame. The report frame includes a report to run and a form for parameters. Parameters are used to pass information to the report subsystem at run time. An example of such a parameter would be a sales report that allows a user to specify for which employees information is desired. Figure 3-3 shows the definition of the task_report frame.

The form for parameters is a VIFRED form. It can include validations and other attributes to ensure that the data are properly filled in. The validation check can specify



```
                          Report Frame Definition

Frame Name : task_report                   Created : 05-nov-1988 13:16:18

Usage : REPORT                            Modified : 05-nov-1988 13:16:18

                  Enter values for the Report Writer Call.

Report Name : task_report                 Form : task_report

Output File (Optional) : printer          Report Source File : rep1.rw



File can be                               Command Line Flags :
   terminal - Put report on terminal.
   printer  - Print report.
   filename - Put report in file.



Define(1)  Destroy(2)  Edit(3)  Rbf(4)  Go(Enter)  Sreport(6)  >
```

Courtesy of Relational Technology

Fig. 3-3 Defining an ABF Report Frame

that the value in the field must be contained in a list of values or in a certain table in the database.

The report definition panel also allows the developer to call VIFRED to construct the parameter form. The developer can also call either RBF or the editor to edit a Report Writer file. Finally, the report subsystem can be called to test the report. After the report has been tested, the developer is right back in ABF and can continue development.

The last type of general-purpose panel is the VIGRAPH frame. Like the QBF and report definition panels, this one allows the developer to call VIGRAPH to develop graphs and to call the GRAPH subsystem to dynamically test graphs. The simplest type of application thus consists of calls to the various INGRES subsystems from a main menu. The main menu requires INGRES 4GL code, but the only code that the developer needs to know is the callframe and exit statements. One menu calls another menu and finally, a QBF, Report, or VIGRAPH frame is called.

We have now seen various levels of sophistication available for application development. QBF can be called on a table, or a JoinDef can be defined. VIFRED can then be used to associate a form with the query targets. RBF and Report Writer are two levels of sophistication available for developing reports.

The next level of sophistication is to use ABF. At its most basic, ABF ties all these general-purpose interfaces together with a series of menus. All the skills that users acquired in learning QBF and VIFRED can be used in this next level, the simple ABF application.

For the rest of this chapter, we will look at more sophisticated types of applications. In these applications, the INGRES 4GL code performs more complex operations, such as dynamically changing the appearance of forms, or passing values into called frames.

Simple Form Interactions

INGRES 4GL allows the application developer to read and write to fields on a form. There are three types of fields:

- simple fields
- hidden fields
- table fields

A simple field is a visible field on the screen. An assignment consists of the field and some valid expression. For example, the following is a simple field assignment:

```
if salary > 10000 then
        special = 'NEEDS APPROVAL' ;
endif
```

In this case, if a large salary is entered on the screen, the user is signaled that this employee salary will need special approval. This example also illustrates the use of the if statement. The if statement takes a comparison, and if the comparison (known as a boolean) is true, then the commands are executed. Otherwise the command (or set of commands) is skipped.

A hidden field is like a simple field, but it is not visible on the screen. The developer declares the hidden field in the initialize section of the INGRES 4GL code:

```
initialize ( counter integer )
        = {
                    counter = 0 ;
        }
'ADD_DATA' = {
                    counter = counter + 1;
                               /* add code here for appending data */
                    if counter > 10 then
                               message 'Added Records ... Exiting' ;
                               exit ;
                    endif ;
        }
```

This block of code allows a user to execute the *Add_Data* block ten times, and then the application is automatically exited. Note that the user is informed that the application is exiting (note also that the append did not actually occur).

Hidden fields are also useful for keeping track of data retrieved from the database. Let us say the application retrieves employee names from the emp table. Employee name is the unique key for each row of the emp table.

If the user attempts to change the employee name, as opposed to the salary or some other nonkey field, then we will have no way of finding the original row. Often, the developer will store key values in hidden fields; then the update is done based on the value of that hidden field.

This method has an important implication for concurrency. Using hidden fields means that a multistatement transaction is not needed to make sure that nobody else changes the value of that row of the emp table. Instead, several users can retrieve the same row. The data are not locked. If and when a user wants to change the data, the application first checks to see that the row being referred to still exists, and then makes the change.

Allowing many users to work on the same data is in sharp contrast with QBF. In QBF, the entire update operation is one multistatement transaction. If the user goes to lunch or decides not to change data, she has still precluded others from working with the same data. This is an example of why one would use ABF to replicate QBF-like functions, but customized to a particular application's need—in this case, the possibility of long think times in a highly concurrent environment.

The third type of field is a table field, which can have several rows and columns. The developer thus needs to identify the particular row and column of the table field. The intersection of a row and column is known as a cell, which is equivalent to a simple field. An example of an assignment for a table field would be:

```
table1.flag = 'YES'
table1.flag[10] = 'YES'
```

The first form of the assignment statement puts the value "YES" into the FLAG column on the current row. The current row is the row the cursor is positioned on. The second version of the assignment statement explicitly assigns data into the FLAG column on the tenth visible row of the table field. In a later section of this chapter, we will examine operations that work on several different rows of the table field.

In addition to putting values into fields on a form, the developer is able to perform several other operations using the Forms Run-Time System (FRS). One common operation is to prompt for user input. The prompt statement displays a string on the screen, takes the resulting user input, and puts it into a field.

An example of a prompt would be to make sure that an operation that is destructive to data is really desired:

```
hidden := prompt 'Are you really sure?';
if hidden = 'Y*' or hidden = 'y*' then
        /*
                    delete the data
        */
else
                    resume ;
endif ;
```

Normally, the prompt command is displayed on the menu line of the screen. The menu disappears and the prompt message is displayed. When the user has entered data, the prompt goes away and the menu reappears.

Another form of the prompt command allows the prompt to be displayed wherever the cursor is currently located. This is known as a pop-up prompt. It is also possible to specify exactly which column and row of the screen the pop-up will appear on.

One possible use of positioning the pop-up prompt is to put the prompt next to the affected field. This is particularly useful in the case of a table field with many different rows of data. We will learn in the section on dynamic applications how to determine exactly where the cursor is or on what row of a table field the relevant data are displayed.

In addition to the prompt command, the preceding example illustrates the use of the *resume* statement. The resume statement puts the cursor back on the field that it was on previously. It is also possible to explicitly position the cursor on another field. For example, a field activation might use the *resume next* command to put the cursor on the next field.

The resume next command allows a field activation to be transparent to the user. The user tabs off the activated field, a block of code is executed, and the cursor appears on the next field. In a normal operation, the user doesn't even know that the code was executed. Only if the developer wishes does the user know something has happened. Thus, if the data are valid, nothing appears to happen. If they are not valid, the cursor goes back to the invalid data, and a prompt is displayed for a valid value.

To communicate with the user, the developer uses a *message* statement, which displays text on the screen. Like the prompt command, the message can go on the menu line or on a specific location on the screen. The message is erased from the screen as soon as the next INGRES 4GL command that affects the screen is executed. On most computers, this is a very short period of time. The friendly application developer would thus add a sleep command after the message to allow the reader time to examine the message:

```
message 'The following is not a valid value '+emp ;
sleep 2 ;
resume ;
```

Message commands are also very useful when a subsequent series of commands will take a long time to calculate. For example, the developer may take a series of numbers in a table field and pass them down into a procedure that performs a matrix inversion. Since this may take a while, it is considered good programming etiquette to put a message on the screen, such as:

```
message 'Processing Request ...' ;
callproc long_time ;
```

The last set of simple FRS commands is for the help system, discussed previously. None of the commands discussed so far have provided any link to data in the database. The next section discusses how data in the database are retrieved into a form.

DBMS Expressions

The right-hand side of an assignment statement is known as an expression. The expression, such as a string constant or the value of a field, is assigned to a field on the form. A DBMS expression is one that moves data from the database onto a field.

A DBMS expression is very similar to a simple expression. Data are retrieved and put into simple fields, hidden fields, and table field cells on a form. However, the DBMS retrieval has the potential to retrieve several rows and thus is treated differently than other assignment statements.

All normal DBMS expressions can be used in INGRES 4GL. Values from the screen can be used in these expressions. For example, we might want to assist users in creating tables. A form can be created having a table name, three simple fields for column names, and three simple fields for the data type of those column names. Note that this is not necessarily the optimal table utility; it would be preferable to allow more than three column names. A more sophisticated table utility would require table fields, however, which are discussed in a subsequent section.

The programmer would, of course, use VIFRED validations to make sure that data types entered by the user are valid and the column names are not longer than INGRES allows. Then, the following block of code would be associated with the form:

```
'CREATE' = {
            message 'Creating Table ...' ;
            create table :table_name (
            :col1_name = :col1_type,
            :col2_name = :col2_type,
            :col3_name = :col3_type ) ;
            resume ;      }
```

Notice the use of the colon in front of the names of fields. The colon tells the FRS to first substitute the values contained in the field into the create statement before passing the completed statement to the data manager. Without the colon, the table created would have columns "col1_name," "col2_name," and "col3_name" instead of the values entered by the user. Not to mention the fact that "col1_type" is not a valid data type and would result in an error message!

An update statement is another example of a database statement. The following code updates the emp table with new salaries and manager names:

```
update emp
          set salary = :sal, manager = :manager
where idnum = :idnum ;
```

Note that the form may have many additional fields. Presumably, this form would have the name of the employee, unless we have personnel officers that recognize all employees by ID number. Since the employee name is not affected, there is no need to include that data in the update statement.

In this example, the field "IDNUM" would probably be set to a display only characteristic in VIFRED. This is because IDNUM is being used as the key to the data. If IDNUM changes, some other employee's salary and manager would be changed. If it is possible to reassign ID numbers, it would be necessary to store the old value of IDNUM in a hidden field, OLD_IDNUM:

```
update emp
          set idnum = :idnum,
          salary = :sal,
          manager = :manager,
where idnum = :old_idnum ;
```

Notice that IDNUM has the same name in the database as it does on the form. INGRES knows which one is being referred to by the context of the name within a particular command. In this case, the set clause of the update requires that a database column be made equal to a value. Thus, the left-hand IDNUM must be a database column; the right-hand IDNUM must be a value—in this case a field on the form, since IDNUM is preceded by a colon.

A retrieval of data can get several columns and many different rows with one statement. Because of this, the select statement requires special handling in INGRES 4GL.

The target of a select statement is not necessarily a simple field or a cell of a table field. If this were so, the select statement would be limited to single-column retrievals.

The basic form of a select statement is a singleton query, which only retrieves a single row from the database. If more than one row meets the search criteria, only the first row is retrieved. The following is a singleton query:

```
emp_form := select * from emp where emp.name= :target ;
```

Notice that the target of the query is a form instead of a field. Making the form the query target allows multiple fields to be filled in with a single statement. The qualification on this query uses the value in the target field so that only the desired row is retrieved. In this case, if more than one row meets the qualification, the user will not know it.

The selection list (the "*"), looks for all columns in the database table that match field names on the form. If there is a column salary in the emp table and a field salary on the emp_form, then that field gets a value. If the VIFRED developer had called the field on the form "SALARIES," then the data would not be retrieved.

To explicitly assign data, the target list can be expanded. In this case the statement would be:

```
emp_form := select salaries = salary from emp
                    where name = :target ;
```

If the target of this query had been a table field, emp_table instead of emp_form, then all rows of data would have been retrieved. Those rows that don't fit into the visible portion of the table field are kept in buffer space in the application. The user can then scroll up and down the rows in the table field.

Because of this difference, select statements on simple fields and table fields must be contained in two different INGRES 4GL commands. It is not unusual to perform one select for simple fields (the equivalent of a master in a QBF master-detail relationship), and perform another select for an associated table field. Thus, all manager data could be put into simple fields, and a list of all employees for that particular manager could be retrieved into the table field.

To process multiple rows on simple fields requires some form of control over the process. This is done in INGRES 4GL using an attached query. Instead of ending the query with a semicolon, a submenu is attached. This submenu has a series of commands that are executed just like a main menu activation would be.

Two special commands are used within the submenu. The next command brings up the next row of data. The endloop command ends the loop and puts the main menu back on the screen. The following is an example of an attached query:

```
select * from emp
{
        'NEXT' = {
                next ;
        }
        'END'  = {
                endloop ;
        }
        'FIRE' = {
                delete emp where name = :employee ;
                next ;
        }
}
```

Notice that a database statement is contained inside the submenu. It is possible to nest submenus to any level. A submenu can also have an initialization section. The initialize block might retrieve all employees working for a manager (although an easier way to perform this particular operation will be shown next).

A submenu requires user intervention. It may be necessary to process all the rows in a retrieval without user action. Instead of a submenu, it is possible to just put a series of commands within the attached query. These commands are executed for each row of data in the select statement. The developer can explicitly request the next row (or end the loop), or an implicit next will be issued when the last command in the block is found.

When both simple fields and table fields are involved in a query, it is possible to put both select statements together into a master detail query. This query has the following form:

```
manager := select * from manager
emp_table := select * from employee
                where emp.manager = :manager
        {
                'NEXT' = {
                        next ;
                }
                'END'  = {
                        endloop ;
                }
        }
```

Every time the next statement is issued, the next master row is displayed, and the associated detail query is reexecuted. Within the submenu, the developer can change master rows, change detail rows, or perform any other operations on the data.

The final form of the select statement uses a special qualification function to dynamically construct the where clause. If a form is on the screen and a "FIND" menu option is offered, the user might enter the qualification on any field of the form. Without the qualification function, the developer cannot easily construct the appropriate select statement.

The qualification select has the following form:

```
emp_form := select * from emp where qualification
                        ( emp.name = :name, emp.salary = :salary );
```

In addition to deciding which fields form the query, the qualification function is able to separate the query operator from the search criteria. If the user enters *>1000* on the salary field, the qualification function builds *emp.salary > 1000* into the query.

The various forms of DBMS expressions allow most of the functions of QBF to be easily replicated, allowing the developer to put together custom applications that meet the needs of a particular organization. The developer can perform extended checking, can retrieve complex information, and has full control over the user interface.

Parameter Passing and Subsystem Calls

The previous two sections examined operations that work within the context of a single frame. DBMS expressions are the result of a particular activation, along with other operations such as prompt statements contained in the code for the activation. Another common action contained in an activation is to call another frame or object in ABF.

When a new frame is called, it is often desirable to retain information from the previous frame. ABF has no form of global memory, but does allow information to be passed between frames and procedures. In a manager/employee master detail query, the employee is a detail record. For every manager there are several employees.

What if a user wishes to examine all projects associated with an employee? Naturally, the user could add yet another table field to the form with an associated submenu, but this begins to be a very complicated frame. One of the principles of good application design is to try to limit segments of code to a single function. In this case, the employee/projects master detail query would sensibly be put into a separate frame with a form.

It doesn't make sense, however, to make the user retype the employee name on the new form. Instead, the employee name is passed in as a result of the callframe statement. In a manager/employee frame, the following code would be a submenu or menu activation:

```
        'MORE_INFO' = {

            callframe emp_proj (
                                    emp_proj_f.employee = :emptable.emp ) ;
                    }
```

This syntax signifies that the emp column on the current row of the emptable table field is passed into the emp_proj_f form of the new frame. This form has an employee field that contains the name of the employee.

Since the nature of the operation is known, it makes sense to retrieve the information for the user without requiring activation of a FIND operation. Thus, the following code would be put in the INGRES 4GL associated with the emp_proj frame:

```
        initialize = {
                        proj_table :=
                                select * from projects
                                where
                                projects.employee = :employee ;
            }
        'DONE' = {
                        return ;
            }
```

The user is presented with a list of all projects in a table field and can scroll up and down this table field and examine nondisplayed rows. When the user is done, she picks the DONE menu item, which returns her to the previous frame. The previous frame contains the data just as it was when emp_proj was called.

Callframe statements can be nested to any depth. Every time the user returns to the previous frame, the data on the form are in the state that they were when the callframe statement was issued. For this reason, it is illegal in ABF to call a frame that is already on the stack, or to use a form that is associated with a frame that is on the stack; to do so would damage the state of the form.

It is also possible to return information to the calling frame. Consider, for example, a sales forecasting application. There is a main frame with the names of all employees and the forecast for their sales. To aid in forecasting, there is a special forecasting frame that pulls up historical data, performs calculations, and uses other aids to pull numbers out of the air. At the end of this forecasting process, the user wants forecast value automatically filled in on the original form.

There are two pieces of INGRES 4GL code. The first piece is in the main frame:

```
'FORECAST' = {
                forecast := callframe fore_frame ;
         }
```

The following piece of code would be in the second frame, the forecast frame:

```
'DONE' = {
                return :answer ;
        }
```

In addition to passing in a single value, it is possible to pass a query into the called frame. The first frame might be used to construct the criteria for a where clause, then pass in the query:

```
callframe newframe (newform = select * from projects
                          where qualification
                          ( projects.num = :num ) ) ;
```

This example takes the value of the *num* field on the current form and uses it to build a select statement on the projects table. The select statement is passed into the frame newframe, which has a form called newform. Notice that no fields are explicitly mentioned in the select statement. This means that any columns in the database that match a field on newform are retrieved. Any fields on the form whose names do not match a database column, and vice versa, are not touched.

It is also possible to call a frame that has a pop-up form associated with it. Pop-up forms and passed queries are very useful for presenting users with lists of possible values. The frame is called in pop-up style. The form on that frame contains a single table field. A query is then passed into that table field. The user positions the cursor on the desired row, and the value of the table field is passed back as a return value.

Frames are one type of object that can be called. Another type is an INGRES subsystem. To call VIFRED, for example, the developer simply issues a *call vifred* statement. Thus, users can be given the opportunity to customize their own forms while executing an ABF application.

Another form of a call statement is to the operating system. The following command on a VAX would issue a call to the mail system:

```
    'EMAIL' = {
                    call system 'mail' ;
                    resume ;
            }
```

When the "EMAIL" option is picked, the screen clears and the user sees the VMS mail package (or whatever mail package happens to be installed on the computer in question). When the user exits the mail system, the screen clears again and redisplays the original INGRES form. This is an example of using ABF as a user interface for non-INGRES applications, or mixing INGRES and non-INGRES into one common user interface.

Embedded Objects: Table Fields

Table fields are a special case in INGRES 4GL because of their ability to display several rows of data. Previously, we saw how a single select statement can be used to load a table field. The looping for simple fields using a submenu was not necessary for table fields.

When data are retrieved from the database, it is possible for more rows to be retrieved than can be displayed on the table field. A table field has two pieces associated with it:

- a data set
- visible rows

The data set is all the data in the table field, including rows that have been deleted and those not being displayed. The visible rows are those currently being displayed. When the user uses arrow keys to move up and down the table field, FRS scrolls up and down the data set displaying as many visible rows as possible. If the user tries to scroll past the end of the data set, an "out of data" message is displayed.

A select statement for a table field simply adds a series of rows onto the current data set. The *inittable* command reinitializes the data set, clearing all rows. In the course of processing, a row can be deleted from view using the *deleterow* command, or a new blank row can be inserted at the current cursor position using the *insertrow* command.

After a select operation, the user could perform a variety of operations on the data. Retrieved data can be updated or new rows can be added. It is also possible to delete rows with the deleterow command.

A special *unloadtable* command is used to process a table field. Like an attached query select statement, the rows are unloaded one by one and a block of code is executed. Two special variables are available for each row of the table field. The first contains the row number in the data set and the second contains the state of the row.

The state variable indicates if a row is new, old and unchanged, old and changed, or deleted. Depending on the value of the state variable, the user can perform the appropriate action on the database: ignore unchanged rows, update changed rows, append new rows, or delete old rows.

If the key value of data has changed, the developer needs to keep track of the original value in order to perform the update operation. In the initialize section, the developer can declare a hidden table field column, just like a hidden simple field is declared. When the select statement is issued, it has the following form:

```
emp_table :=
        select   emp_table.hidden_emp = emp.name ,
                 name = emp.name,
                 salary = emp.salary
        from emp ;
```

Notice that the name column from the database is being loaded into two fields on the form. The visible field is available to users for update. The hidden field keeps track of the key values so that the original row can be found in the database.

Finally, a special-purpose *scrollto* function is available to scroll the table field to the appropriate row on the data set. This could be used to provide a "FIND" option for the user or to scroll to a particular row number.

A table field is an object that contains objects, just as a form contains fields. The table field has certain properties associated with it, such as the data set. It has certain special-purpose operations, such as the unloadtable command. In turn, the components of the table field also have certain attributes and operations associated with them.

A table field is an example of a complex object: several different simple fields operating in a coordinated fashion. We will see in the next chapter that University of California researchers are working on more complex forms of table fields. Instead of containing simple fields, this research project is able to embed other objects, such as images, inside of the table field format.

Dynamic Applications

A variety of mechanisms are available in INGRES 4GL that allow the application developer to dynamically change characteristics of the FRS. For example, the visual attributes of a field can be changed from the normal display to reverse video blinking. If the field has no visible title and is display only, the user does not even know it is there.

When sensitive data are retrieved from the database, the developer can put the word "SECRET" into this display-only field. Then, the visual attributes of the field can be changed to reverse video and blinking. Although this might not be considered highly aesthetic from a user interface design standpoint, it does highlight the sensitive nature of the data.

A dynamic application is thus one that is able to adapt the user interface to the particular user running the program. QBF is an example of a dynamic application. The user can pick an arbitrary table, and a form is dynamically constructed for that table. The user can choose an arbitrary query and an SQL statement is dynamically constructed to receive the data.

In addition to setting the characteristics of the visual display, the developer can find out the status of the FRS. This is done with an *inquire_frs* command and a target object. These objects include:

- the FRS
- a form
- a simple field
- a table field
- a row in a table field
- a column in a table field

The FRS object is used to get information about the overall status of the user interface. Types of information include FRS errors, the current terminal type, and current mapping file. With an set_frs command, the mapping file can be changed.

The developer can also examine and set the mapping between an FRS function and a key. Thus, the developer could inquire as to the current mapping of a certain control key. If that key is reserved for an important function, the developer would then look for another control key. Once an unused one is found, it could be mapped to an FRS command such as help, and a message would be displayed on the screen to use that key for help information.

The developer can also inquire about the current position of the cursor on the screen and the size of the screen. Then a prompt, message, or pop-up form can be displayed at the appropriate location, taking into account the size of the workspace. This capability becomes increasingly important as users move to a workstation environment, because windows on a workstation can be resized at any time by the user.

The developer can also inquire what user action caused an activation to occur. A field activation, for example, occurs whenever the user leaves a field. Leaving a field could be the result of a cursor key, a forward tab, or a backward tab. Depending on which action occurred, a different block of code could be executed.

Information about any form that has been displayed can be retrieved, including the name of the form, the size of the form, and the field that the cursor is positioned on. Using this information, the developer can determine where on the previous form the user was when the current frame was called. Depending on the cursor position, different types of information might be retrieved.

Information about fields includes the name, data type, and various visual attributes. It also includes a change indicator that is set whenever the field has new data entered. If the data have not changed, there is no need to replace them in the database.

Table field information is available for the whole table field or for individual rows or columns. For the table field itself, the developer can inquire about the name and size of the table field and can also look at the current row that the cursor is on, the size of the data set, the number of displayed rows, and the number of deleted rows.

Individual rows can be examined for the change variable as well as for the position of that row on the screen. The position is useful for displaying pop-up objects, such as forms or prompts, at the appropriate position.

Columns in a table field are simply cells of the table field; they have the same types of attributes as simple fields. It is possible to look for the current cursor position and then dynamically change a desired cell to reverse video in order to highlight a particular item in the table field.

In addition to changing portions of the form, the developer can set the display mode of that form. A read only form allows users to examine data, including scrolling up and down table fields, but does not allow users to change any data. A form in query mode is used for the qualification function so that the FRS can break up a query field into the query operator and the search value. Forms in fill or update mode allow the user to enter and change data.

The mode of a form applies to all the simple fields on a form. Table fields can have a different mode set. This is useful to make the simple fields in fill mode and the table field in update mode, allowing users to change detail records but not to change the master information. Finally, the developer can issue inquire_ingres statements to examine the status of the database. Information from this command shows the total number of rows retrieved, as well as error messages and error text.

It is important to examine error messages even in a well-designed application. One type of error message is the deadlock indicator. Deadlock (discussed in Chapter 6) is an inherent part of any concurrent environment; failing to check for this condition means that a user might think that data were updated in the database even though the transaction was aborted.

Procedures and Embedded 4GL

Several types of frames have been examined. QBF, Report, and VIGRAPH frames are general-purpose frames requiring no programming. User frames couple a form with some INGRES 4GL code, which can be as simple as defining a menu that calls QBF, or can be quite complex, consisting of various DBMS and FRS operations.

An INGRES 4GL procedure is an object in ABF that does not have a form associated with it. Here, we use INGRES 4GL as a general-purpose programming language. A procedure declares variables and then issues a series of commands. These commands can use submenus and prompts to interact with the user, but the form on the calling frame remains displayed.

Procedures are useful for operations that are frequently repeated. A frame cannot be called again once it is on the calling stack because it has a form associated with it. A procedure can be called as many times as necessary. When a particular database operation is repeated many times, it is frequently put into a procedure. By putting the keyword *repeated* in front of the operation, the data manager is informed that this operation will be executed again. The data manager saves the query execution plan (discussed in Chapter 5), allowing previous iterations of the operation to execute significantly faster.

Another type of procedure is the Embedded 4GL procedure. This type of procedure is written in a third-generation language, with SQL or FRS embedded in it. If the application performs statistical processing, such as a moving average of historical data for forecasting, INGRES 4GL is probably not the appropriate language. A third-generation language can be used to perform this particular function and return the forecast results to the user.

Embedded 4GL allows the developer to put SQL and form manipulation statements into a third-generation language. Instead of using files to access data, the services of the database, such as a high-level data manipulation language and the protection of data integrity, are used. Data are moved from program variables into database tables and columns instead of into files defined and manipulated by the programmer.

It is also possible to write Embedded 4GL programs as stand-alone programs, not using the services of ABF. Normally, ABF would handle the details of linking in the appropriate libraries. With a stand-alone Embedded 4GL program, the programmer develops a program using a text editor, and then specifies the details needed to compile and link all of the modules together.

It is often tempting for programmers new to the INGRES environment to write all their applications in Embedded 4GL and not use the services of ABF. The reason stated for this development method is usually performance. There are times that this is a valid reason, but usually it is not. When INGRES 4GL code is compiled, it is turned into C code. Compiling INGRES 4GL is like writing a large program. Many functions that require many lines of C code are done in a single line of INGRES 4GL. While it is possible to write the function in a more efficient manner using C instead of INGRES 4GL, the goal in most organizations is not necessarily just limited to more efficient C code.

The drawback of this development philosophy is that writing third-generation language programs takes longer than writing INGRES 4GL programs. Granted, the machine may run more quickly, but the programmer runs more slowly! In most organizations, the programmer is the scarce resource, not the machine. It is fairly straightforward to upgrade a machine to run quicker; it is significantly harder to hire more programmers or to upgrade the skills of existing programmers.

Even though an Embedded 4GL program takes longer to write than a straight INGRES 4GL program, there are some instances when this type of development is necessary. Highly critical portions of code are one example; another is functions not performed by the INGRES 4GL, such as statistical analysis.

A third reason to use Embedded 4GL is to take advantage of its dynamic capabilities. Both SQL and FRS operations can be dynamically constructed and executed, allowing very general applications to be developed that allow the user to pick a query target at run-time. The program then issues the appropriate calls to work with a form or an SQL statement to retrieve data from a database table.

QBF is an example of such a dynamic application; it works on any object in the database. At run time, QBF determines the characteristics of the form and dynamically issues a query; it then dynamically determines where to put the resulting data and displays it to the user. Note that replicating QBF functionality for most applications does not require dynamic SQL or FRS because the characteristics of the forms and the database are known by the application developer. For these special-purpose situations, INGRES 4GL and ABF are perfectly adequate.

Dynamic SQL is often used in preparing an interface from another software system into INGRES. Chapter 9 discusses a variety of heterogeneous user interfaces, such as spreadsheets, that are able to use INGRES as a data repository.

Dynamic SQL statements are prepared in the programming language, then dispatched to the back end for execution. For example, the user of a spreadsheet may wish to retrieve data from the database. The spreadsheet would first display a list of available tables by querying the system catalogs. Then, after the user has made a choice, a select statement would be sent off to the back end to retrieve the data.

Often, it is not known in advance how much data will be returned by a particular statement. Dynamic SQL allows a description of the incoming data to be received first. This description is examined by the program, and appropriate data structures are set up to receive the data. Then, the data are received a row at a time using a cursor.

Dynamic FRS is very similar to dynamic SQL. The programmer can receive a description of a form and set up program variables to correspond to the fields on a form. Then, FRS statements can be used to interact with the form. For example, the program could establish a series of activations. When a menu item is picked, the program would examine the fields to find out which ones had data in them. The data would be read into program variables and a query constructed that interacts with the database.

Embedded 4GL is the most sophisticated level of access to an INGRES database. These tools are used by programmers to develop tools such as links to other subsystems. While ABF and all the other INGRES subsystems can be used by nonprogrammers, Embedded 4GL requires some training in programming.

Image Execution and Construction

When an image is created in ABF, a number of different operations takes place. Each of the INGRES 4GL files are preprocessed to generate C code. These C programs are then compiled to turn them into machine language. All forms are also compiled to turn them into machine language.

All of these different components are then linked together into a single executable image or program. Included in this linked image are different libraries used by INGRES to access the FRS, the data manager, and other functions.

One of the prime advantages of ABF is that it coordinates image construction and does it transparently. The application developer does not have to specify all the different pieces making up the completed program. When a new version of the application is compiled, ABF is able to determine which files have changed, and only recompile the portions that are different. All of the pieces are then recompiled into an executable image.

Once the application is constructed, users are able to run it as they would any other program. The developer moves the executable image to some central location, and defines a symbol used to start the application. On VMS, the symbol might be as follows:

```
go := $emp_app.exe main_menu
```

When the user types "GO," the EMP_APP image is activated and the frame called "TOPFRAME" is displayed on the screen. Note that different symbols can be defined for different users, each one having a different beginning frame. This allows multiple points of access to one application.

When the application is actually run, a wide variety of operations takes place for the user. First, the image is loaded into memory and channels are opened for input, output, and errors. Then, exception handlers, which are used to trap interrupts and deal with errors, are initialized. Next, the FRS is initialized.

When the FRS is activated, it first checks the attributes of the terminal, such as the baud rate. Then it finds the type of terminal and goes to the terminal capability file to find out how to perform physical operations on that device. Next it reads in all mapping files to determine how to map logical functions to physical functions.

After the initial housekeeping is done, the ABF Run-Time System takes over. First, it loads a table into memory with the names of all frames and procedures in the application, any forms associated with them, and information on what languages the procedures are written in. The language information is necessary so that parameters are passed in the right format.

For each frame called, ABF first checks to see if it is a subsystem call, such as QBF or a user frame. For user frames, ABF first adds the form into memory, either from a compiled file or from the database if necessary. Then, it initializes all table fields and allocates memory for their data sets.

If any parameters were passed into the form, ABF then maps the parameters to the fields on the form. Next, it runs any queries that were passed in. Finally, it executes the initialize block of the frame. After the frame is initialized, ABF goes into a display loop. The display loop allows various types of forms movements and performs any appropriate VIFRED validations. ABF also monitors the user for any activations.

When an activation occurs, it passes control to the block of code associated with that activation.

Finally, when the application exits, it drops the forms system, which frees up memory and cancels the session with the data server. It then exits the application, possibly with status messages.

All of these operations are done without the knowledge of the user. The user sees a form, fills in values, and picks various menu operations. All of the functions are also done without any explicit programming by the application developer—the developer simply defines frames and writes INGRES 4GL code.

Summary

ABF allows simple and complex operations to be developed without a great deal of programming knowledge. The developer can take advantage of the building blocks in the general-purpose user interfaces and incorporate them into an application.

An application can thus consist of simple calls to INGRES subsystems. These subsystems use reports, JoinDefs, tables, graphs, and other objects. An important characteristic of this development environment is that it uses the same concepts seen throughout the other general-purpose subsystems. The developer is able to begin acquiring simple skills, and the concepts acquired in these simple systems are useful in more sophisticated environments. This is in contrast to some other systems that require the user to relearn a new set of skills every time she moves up to a new level of sophistication.

The application development environment has two important characteristics. First, the interface is object-oriented. Object-oriented interfaces allow functionality to be increased without increasing complexity. Objects each have their own characteristics and can be incorporated into higher-level objects.

The second characteristic of the user interface is a tight integration of the different components. Forms, for example, are used throughout the system—for ABF, QBF, RBF, and Embedded 4GL programs. Most of these components, such as forms, have defaults. A default form can be built and quickly incorporated into a system. Later, more complex versions of the form that change the default settings can be defined.

ABF is the environment used to integrate the various objects of INGRES into completed applications. Objects can be data objects, such as views, JoinDefs, or tables. Objects can also be display objects, such as simple fields or table fields. A complex display object is the form or table field containing simple fields. A graph is another example of a simple object. The report is another example of a display object.

INGRES 4GL is the language used to integrate these various objects together. The INGRES 4GL includes the following capabilities:

- query/update the database
- simple and DBMS expressions
- fine tune cursor placement and set field attributes

- define activations
- conditional processing within an activation
- error messages
- calls to procedures, frames, subsystems, and the operating system
- passing parameters
- hidden fields
- submenus
- selective processing on table fields

These capabilities are general enough to apply to most applications. Occasionally, it is necessary to move beyond the capabilities of INGRES 4GL into lower-level programming languages. Embedded 4GL is a language very much like INGRES 4GL; it operates on forms and on the database. Embedded 4GL allows third-generation languages such as COBOL, C, or FORTRAN, to be integrated into the database environment.

Chapter

4

Extensions to the User Interface

Workstations vs. Terminals

The INGRES Forms Run-Time System (FRS) can function effectively on a wide variety of different terminals, ranging from IBM 3270 display stations, to DEC VT series smart terminals, to dumb terminals. Because INGRES also runs on a variety of different hardware platforms and is accessible across a variety of different networking technologies, applications are highly portable.

Portable applications allow users on different terminals to see the same user interface. That interface is composed of a form consisting of trim, simple fields, complex table fields, and a menu of operations that the user can execute. While this interface is ideal for a terminal-based computing environment, it does not take full advantage of the power of workstations with graphical user interfaces and a mouse input device.

A variety of efforts are underway to move the INGRES tool set into a workstation-based environment characterized by windows, icons, and other more intuitive methods of interacting with an application. This chapter discusses two of these efforts.

First, Simplify is discussed. Simplify supports a collection of decision support tools on a Sun Workstation using the Open Look windowing environment. This product allows users of a Sun Workstation to take full advantage of the iconic interface available on workstations and to access data in an INGRES database. Second, the Picasso research project at the University of California at Berkeley is described. Picasso is an integrated environment for the development of end-user database applications. Picasso is under the direction of Professor Lawrence A. Rowe, a founder of Relational Technology, who was one of the original developers of the INGRES forms user interface and tool set.

While Simplify is a supplement to end-user tools such as QBF, the Picasso project attempts to change the nature of the application development tools that work with a database. The features of the Picasso project will not necessarily appear in future versions of INGRES, but the research does point out some important trends in the way developers are reconsidering how data are accessed and presented.

Workstations have several important characteristics that differentiate them from a more traditional terminal-based user interface. First, the user has a personal, powerful computer. This computer typically consists of a 2-10 MIP CPU, several megabytes of memory, and a powerful bit-mapped graphics screen. Second, the workstation is almost always participating in a network, which allows different programs running on other computers in the network to be accessed from the workstation. In the case of INGRES, a front-end application on a workstation can access data repositories throughout the network.

Third, the user interface on the workstation is graphical. Instead of hitting a function key to perform a function, the user selects a menu item on the screen using a mouse or other pointing device. This option may have submenus, and the user moves the mouse down to the desired submenu options and clicks the mouse button again to select the option.

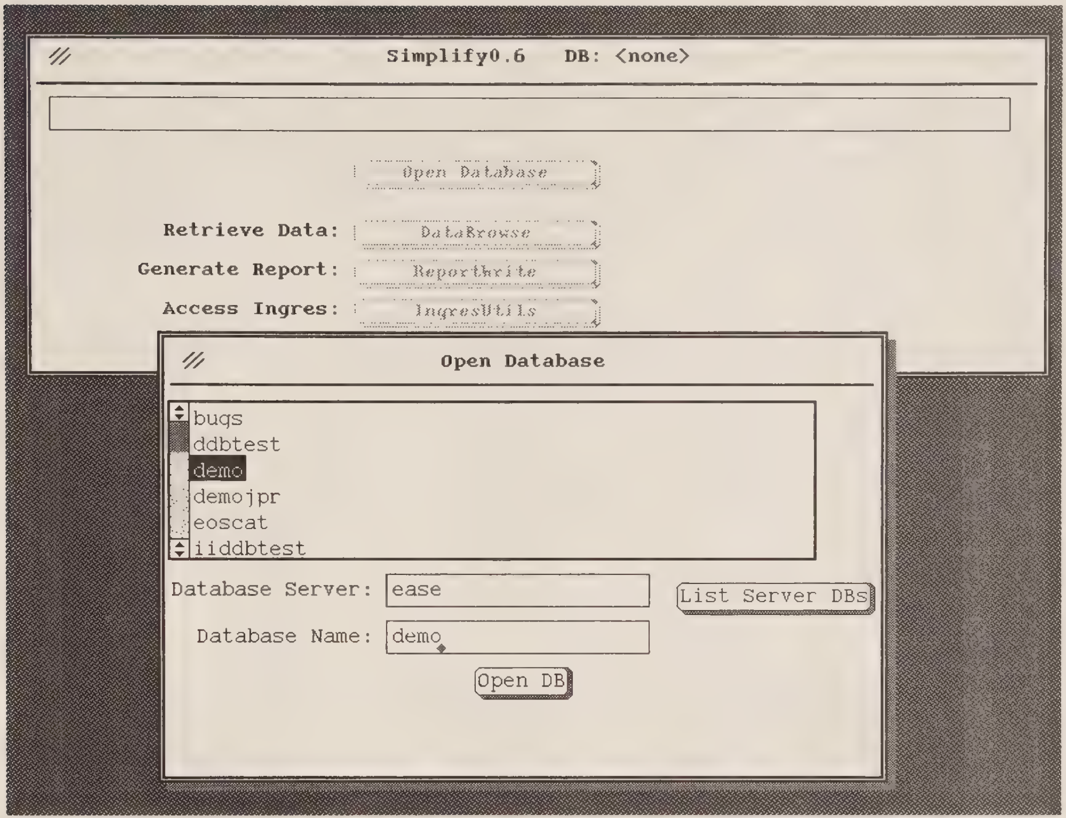
Another aspect of the graphical interface is that a user has multiple windows open on the screen at the same time. Windows are used to run different applications or to display different interfaces or views of a single application. For example, Figure 4-1 shows two windows used in Simplify. The first window gave the user the option of opening a database or performing various Simplify utilities.

When the user pointed to the "OPEN DATABASE" option, another window opened up, listing the available databases. The user can interact with this window in a variety of ways. He can enter a database name in a text field or point to a particular database name and click a mouse button to select it.

Any of these windows can be closed by pointing to the hash marks in the upper left hand corner of the window. A closed window is represented by a small icon that indicates what application that window is running. Figure 4-2, for example, has a small mailbox icon in the left corner that represents a mail application (see page 90). When new mail arrives, the little flag on the mailbox goes up. When the user points the mouse at the mailbox icon, the window opens and the user can read her mail.

Simplify

Simplify is a product originally developed by Sun Microsystems as a graphical interface to database systems. Relational Technology and Sun Microsystems, under a joint development project, have enhanced the original system to meet the needs of end users for graphical decision support tools.



Database Server: ease

Database Name: demo

Courtesy of Sun Microsystems

Fig. 4-1 Opening a Simplify Database

Simplify has many important features that illustrate the direction that data access is taking in a workstation-based environment. First, Simplify is well integrated into the other applications that run on a Sun Workstation.

Integration has two aspects. First, there is a common look and feel standard, meaning that all the different applications, including, for example, a word processor, a drawing package, or a database package, function the same way. Integration also means that it is easy to move data between different applications.

Look and Feel Standards

A primary advantage of INGRES applications is a common look and feel on terminals. Help is always the same key in all subsystems. Moreover, the forms system is the same so that fields function the same way, in VIFRED, QBF, RBF, and ABF.

The implication of a common look and feel standard is that users are able to use previous skills when they move to a new application. They are not forced to switch between different help keys or to consult a manual to find out where the help key is located. Users can concentrate on the substantive portion of their work and not on having to constantly relearn or remember the procedural aspects of interacting with the computer.

In a window-based environment, a common look and feel standard becomes especially important. This is because a workstation often has a wide variety of different applications all active at the same time. Often, the windowing environment allows the user to move data between windows. Data retrieved from an SQL query could thus be copied from the ad hoc query interface into a document controlled by a word processor for further formatting.

Since Simplify runs on Sun Workstations, it conforms to the Open Look look and feel standard. This standard has been endorsed by several vendors, including Xerox and AT&T. Once a user learns the Sun windowing environment, very little additional training is necessary to use Simplify.

The Open Look standard defines what the windows on a display look like and how the user interacts with the windows. For example, a mouse is defined as having three basic functions:

- select objects and manipulate controls
- adjust an object such as resizing it
- select and display menus

Open Look then defines which buttons on a mouse are used for these three basic functions and defines the properties of a menu. When a menu item is selected, for example, there is always a default selection in bold. That selection is in boldface. If a menu is selected by the user, it is highlighted. Open Look also defines standard menu options that should be available in all applications.

It should be noted that, as the case of other areas of the computer industry, there are several look and feel standards. DEC, IBM, and Apple all have different look and feel standards for their different computing environments. One of the challenges for end users and third-party software companies like Relational Technology is moving their applications into a window-based environment that conforms to the look and feel standards of each of the different platforms on which INGRES runs while at the same time still retaining software portability.

DataBrowser

QBF is one example of a visual query method. It allows the user to point to a field and fill in the applicable values. Filling in "> 2000" in the Salary field of a form, for example, is equivalent to the SQL statement:

```
select * from table_name where table_name.salary > 2000
```

QBF is one of several examples of visual user interfaces that generate SQL for the user. Another example is IBM's Query By Example. A third is the Simplify DataBrowser.

The DataBrowser allows the user to store and edit complex queries that are represented graphically and through the user of dialogue boxes. To construct a query, the user first opens the relevant database. Then, the user picks the DataBrowse menu option and requests the Edit query command.

The user is then presented with a list of tables in the database. By clicking on a given table, the query display area shows the columns in that table (see Fig. 4-2), and the columns are displayed on the screen. Next, the user can point the mouse at each of the fields to select them. In Figure 4-2, the user has selected the name, number, and job columns of the emp table for display. A check mark appears next to that column to signify that it has been selected for retrieval.

In addition to the emp table, Figure 4-2 has a second table, the member table. This table signifies that a particular person is a member of an organization. To join the two tables together, the user points to one of the fields in the join and holds down the mouse button. A menu appears, and the user selects the "CREATE JOIN" option. Then, the user moves the mouse down to the second table and points to the emp column and clicks the mouse button.

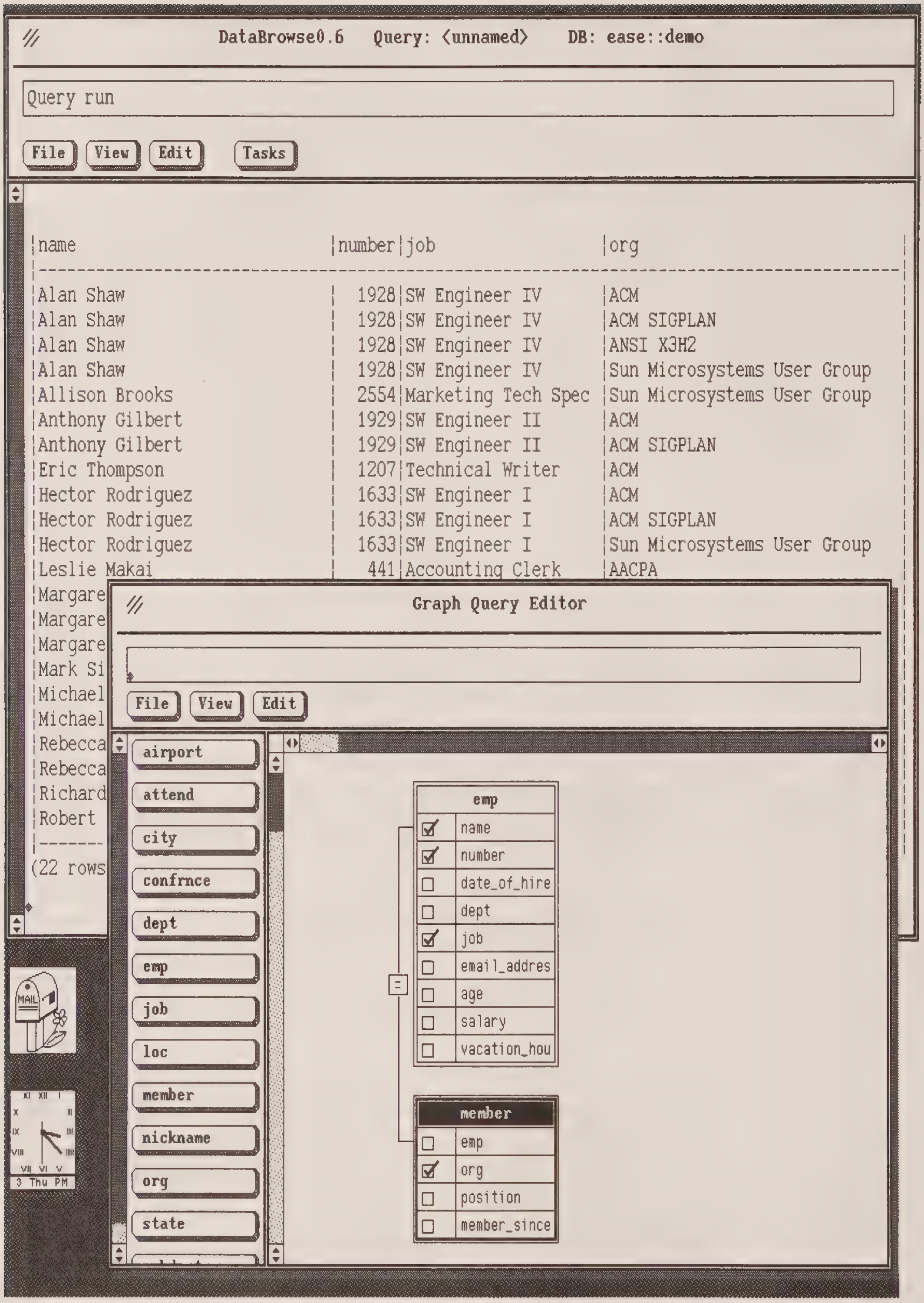
One interesting feature of Simplify is that the database administrator can specify PreJoins. A PreJoin tells the database what fields might be joined between two tables. When two tables are displayed, and a PreJoin exists, the query editor automatically joins the tables. It is of course possible to change the default join in the query editor if the user has a different type of query to run than the database administrator had in mind.

QBF JoinDefs are a similar concept to the Databrowse PreJoins. QBF does a default join by looking for columns with the same name. With a PreJoin, however, the database administrator can specify that two tables be joined across differently named columns, just as the user would in editing the JoinDef definition.

At this point, the query can be saved or run. When the query is run, it appears in the DataBrowse window located behind the query editor. Notice also that the screen has two other windows on it, both in iconic form: a clock and a mailbox. Anytime the mouse cursor moves out of one window to another, control is transferred to the new application. Moving the cursor over to the mailbox icon would allow the user to interact with the mail program.

A more sophisticated query is possible by restricting the results (see Fig. 4-3). This is equivalent to building a where clause. A restriction is done by clicking on a particular field, which is then highlighted. By holding the menu button down anywhere in that field, the user is presented a with a set of options, including the ability to create a restriction.

Creating a restriction first begins by picking an operator. In Figure 4-3, the user is creating a query that only retrieves employees with a number less than 2000. The user clicks on the less than operator and then fills in the value 2000 in the window. Note that the query editor now shows the presence of the restriction in the window. If the



Query run

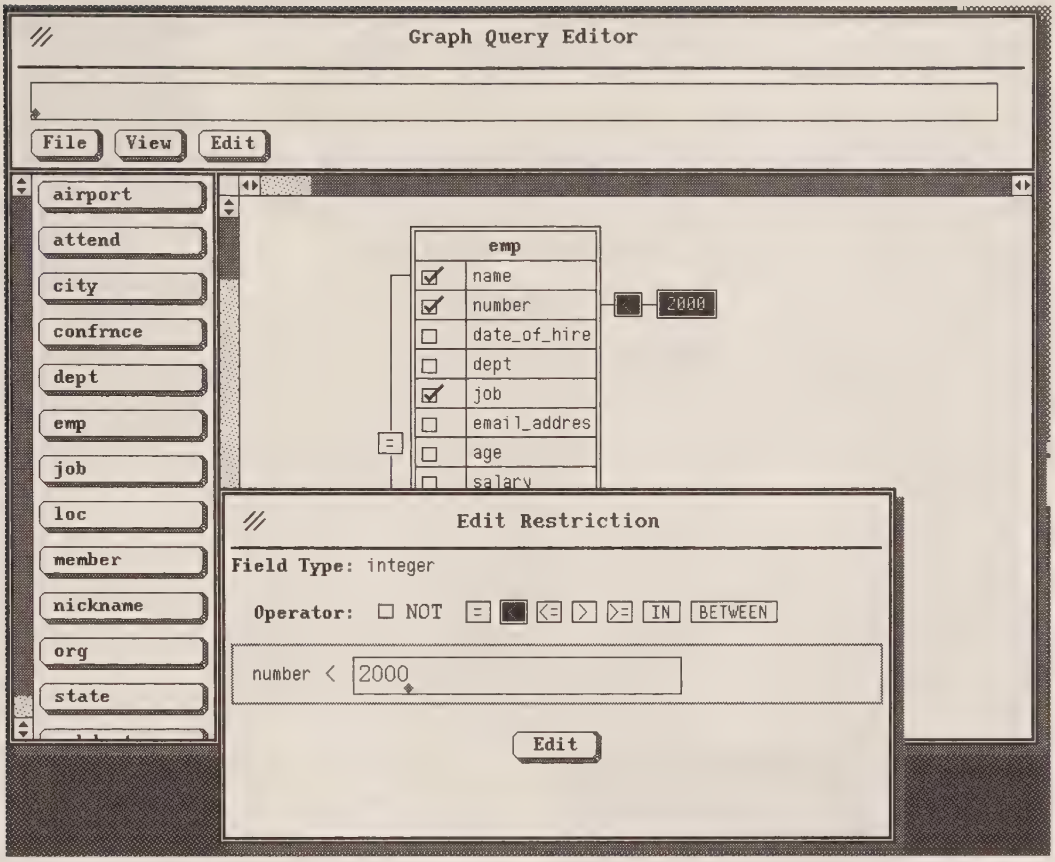
```
|name                          |number|job                 |org                           |
|------------------------------------------------------------------------------------------
|Alan Shaw                     |  1928|SW Engineer IV      |ACM                           |
|Alan Shaw                     |  1928|SW Engineer IV      |ACM SIGPLAN                   |
|Alan Shaw                     |  1928|SW Engineer IV      |ANSI X3H2                     |
|Alan Shaw                     |  1928|SW Engineer IV      |Sun Microsystems User Group   |
|Allison Brooks                |  2554|Marketing Tech Spec |Sun Microsystems User Group   |
|Anthony Gilbert               |  1929|SW Engineer II      |ACM                           |
|Anthony Gilbert               |  1929|SW Engineer II      |ACM SIGPLAN                   |
|Eric Thompson                 |  1207|Technical Writer    |ACM                           |
|Hector Rodriguez              |  1633|SW Engineer I       |ACM                           |
|Hector Rodriguez              |  1633|SW Engineer I       |ACM SIGPLAN                   |
|Hector Rodriguez              |  1633|SW Engineer I       |Sun Microsystems User Group   |
|Leslie Makai                  |   441|Accounting Clerk    |AACPA
|Margare
|Margare
|Margare
|Mark Si
|Michael
|Michael
|Rebecca
|Rebecca
|Richard
|Robert
|-------
(22 rows
```

emp
☑ name
☑ number
□ date_of_hire
□ dept
☑ job
□ email_addres
□ age
□ salary
□ vacation_hou

member
□ emp
☑ org
□ position
□ member_since

Courtesy of Sun Microsystems

Fig. 4-2 A Graphical Representation of a Query



☑ name
☑ number < 2000
□ date_of_hire
□ dept
☑ job
□ email_addres
□ age
□ salary

Field Type: integer

Operator: □ NOT = ■ <= > >= IN BETWEEN

number < 2000

Courtesy of Sun Microsystems

Fig. 4-3 Adding Restrictions to a Query

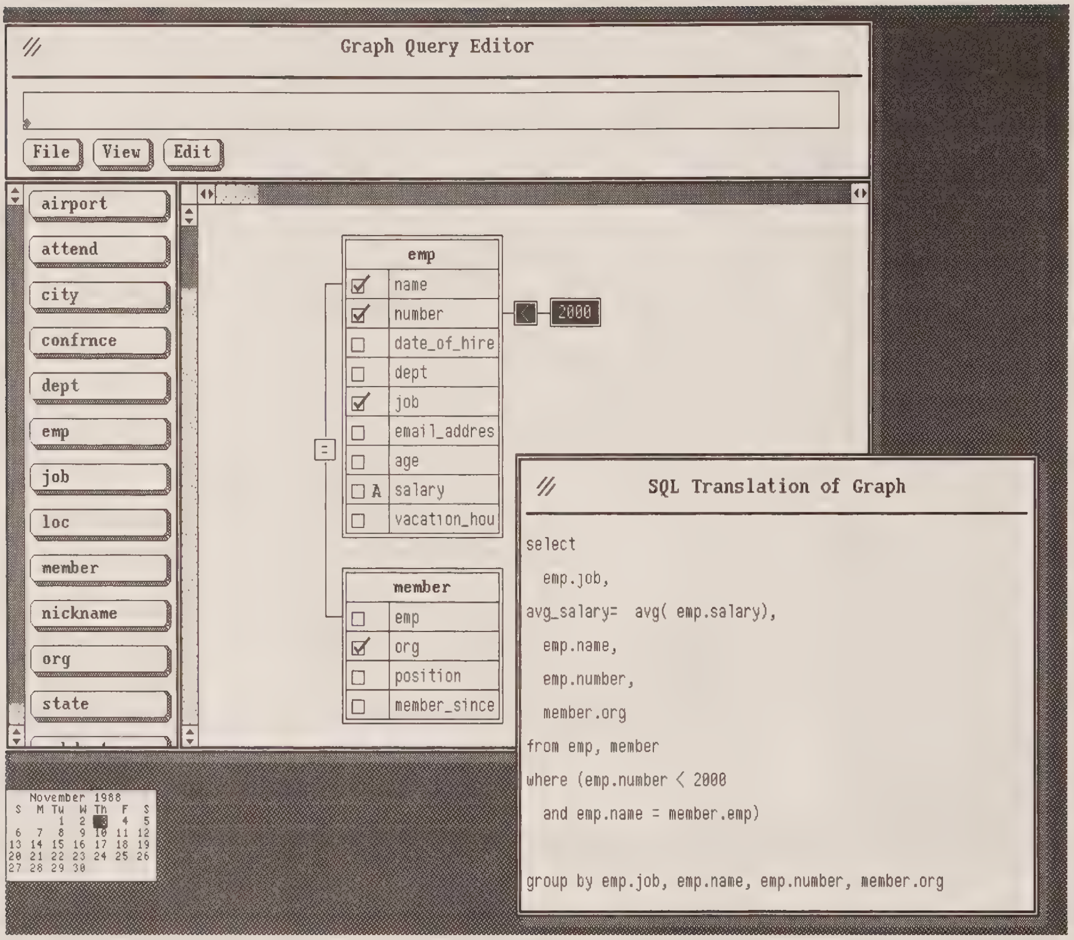
field is also checked, that means it will be displayed. It is possible to qualify a field without displaying it.

As with QBF, the Simplify subsystems generate SQL on behalf of the user. The SQL is sent to the back end for processing and the results are then displayed. Figure 4-4 shows the SQL equivalent of the previous query.

Viewing the SQL equivalent of a query is a useful tool for programmers and others that will be using SQL in INGRES 4GL or Embedded 4GL applications. Simplify can be used as a rapid prototyping environment. The SQL can then be captured and moved over to another window containing the code for the application being developed.

Report Generator

The Simplify Report Generator is similar in many ways to the RBF subsystem. Like RBF, the user is able to point to fields to place them in the proper report format. Be-



```
select
  emp.job,
avg_salary=  avg( emp.salary),
  emp.name,
  emp.number,
  member.org
from emp, member
where (emp.number < 2000
  and emp.name = member.emp)

group by emp.job, emp.name, emp.number, member.org
```

Courtesy of Sun Microsystems

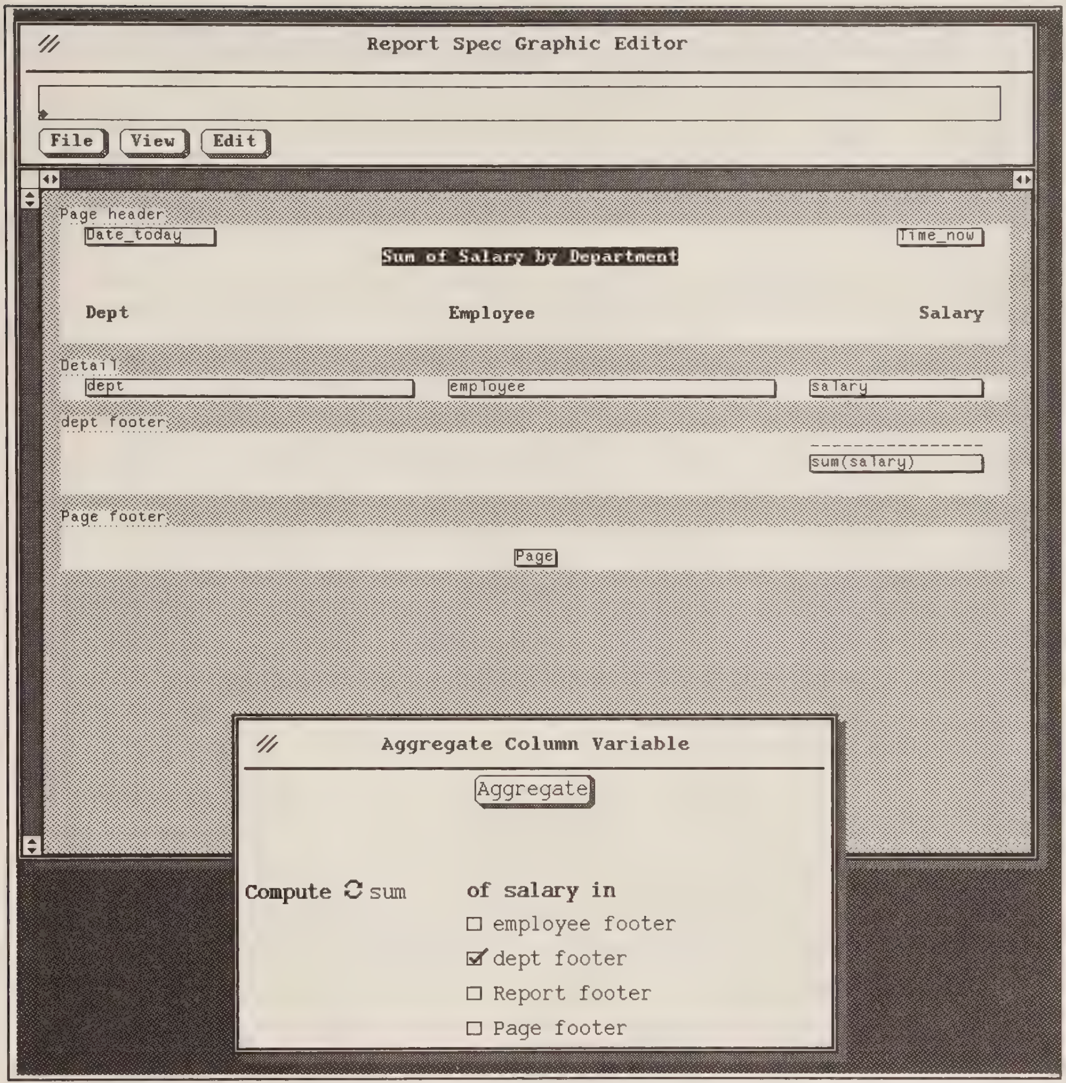
Fig. 4-4 SQL Equivalent of a Query

cause the Report Generator runs in a workstation environment, it is capable of a more intuitive method of developing report specifications than the terminal-oriented RBF.

Reports generated by the Report Generator are stored in two places. First, they are stored in the internal Simplify database. This allows the report to be edited in the Simplify environment. The second place the report is stored is in the INGRES system catalogs. This allows the report to be run by any user, even one that does not have access to Simplify. The report is loaded into the system catalogs using the sreport command.

The report specification process begins by choosing a query that will be used to gather data for the report. The query is specified using the DataBrowse utility and then saved. Sort columns in the query are used for aggregates and header/footer sections for the report.

The Report Generator starts by presenting a default format for the report, just as RBF does. Figure 4-5 shows the editing phase of the Simplify Report Generator. The



Compute sum of salary in

☐ employee footer

☑ dept footer

☐ Report footer

☐ Page footer

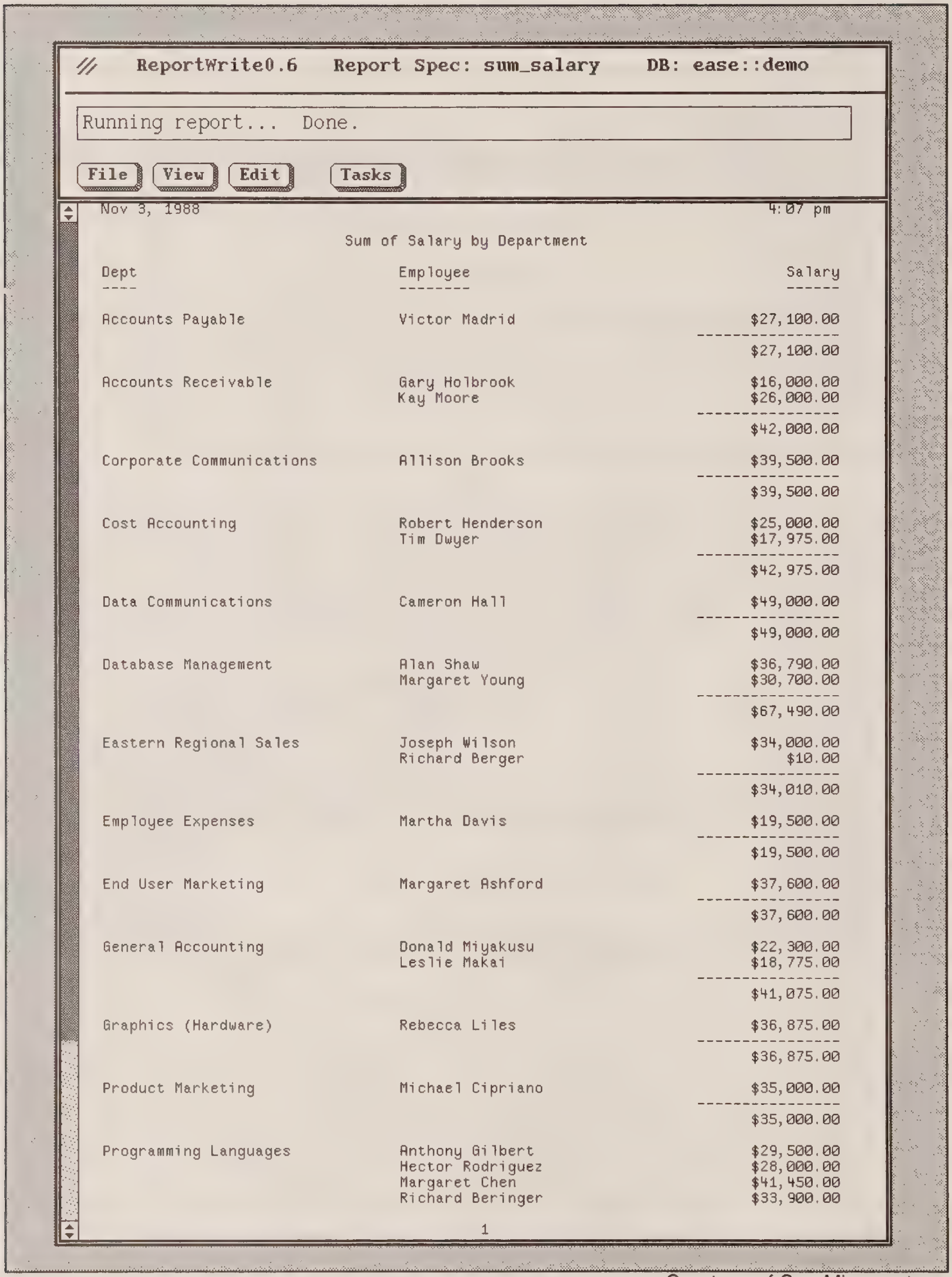
Courtesy of Sun Microsystems

Fig. 4-5 Simplify Report Specification

page header section of the report contains column names, just as in RBF. In addition, the user has added boxes that display the current date and time at the top of each page.

In Figure 4-5, the user has added two footer actions for the report. First, at the bottom of the report is a page footer section that contains the current page number for the report. The user has also added a footer for the department column. In this column, the sum of salary is being computed. Notice that the sum of salary could have been specified for any of the break columns, including the employee and department columns, as well as the report or page footer sections.

Once the report is specified, the user saves it. The report can then be run in a window (see Fig. 4-6) or spooled to a file or a printer. One advantage of the window-based computing environment is that both the report output and the report specifications



```
Running report...  Done.
```

```
Nov 3, 1988                                                          4:07 pm

                           Sum of Salary by Department

Dept                             Employee                              Salary
----                             --------                              ------

Accounts Payable                 Victor Madrid                     $27,100.00
                                                             ----------------
                                                                   $27,100.00

Accounts Receivable              Gary Holbrook                     $16,000.00
                                 Kay Moore                         $26,000.00
                                                             ----------------
                                                                   $42,000.00

Corporate Communications         Allison Brooks                    $39,500.00
                                                             ----------------
                                                                   $39,500.00

Cost Accounting                  Robert Henderson                  $25,000.00
                                 Tim Dwyer                         $17,975.00
                                                             ----------------
                                                                   $42,975.00

Data Communications              Cameron Hall                      $49,000.00
                                                             ----------------
                                                                   $49,000.00

Database Management              Alan Shaw                         $36,790.00
                                 Margaret Young                    $30,700.00
                                                             ----------------
                                                                   $67,490.00

Eastern Regional Sales           Joseph Wilson                     $34,000.00
                                 Richard Berger                        $10.00
                                                             ----------------
                                                                   $34,010.00

Employee Expenses                Martha Davis                      $19,500.00
                                                             ----------------
                                                                   $19,500.00

End User Marketing               Margaret Ashford                  $37,600.00
                                                             ----------------
                                                                   $37,600.00

General Accounting               Donald Miyakusu                   $22,300.00
                                 Leslie Makai                      $18,775.00
                                                             ----------------
                                                                   $41,075.00

Graphics (Hardware)              Rebecca Liles                     $36,875.00
                                                             ----------------
                                                                   $36,875.00

Product Marketing                Michael Cipriano                  $35,000.00
                                                             ----------------
                                                                   $35,000.00

Programming Languages            Anthony Gilbert                   $29,500.00
                                 Hector Rodriguez                  $28,000.00
                                 Margaret Chen                     $41,450.00
                                 Richard Beringer                  $33,900.00
                                         1
```

Courtesy of Sun Microsystems

Fig. 4-6 Simplify Report Output

can be displayed on the screen. If the report does not come out the way the user desired it, he simply moves the mouse back over to the report editing window and changes the specifications, saves the report, and runs it again.

It is a fairly simple matter to take a report and move into a word processing program for inclusion in a document. Once in the word processing program, such as SunWrite, it can then take advantage of all the formatting capabilities of a windows-based word processor.

Moving the report into another application can be done one of two ways. First, the report can be spooled into a file and then imported into the word processor. It is also possible to use the clipboard on the windowing system to move the data. The user draws a box around the desired data, cuts it to the clipboard, moves to the next window, and pastes it into place.

For example, Figure 4-7 shows the report from Figure 4-6 after it has been moved into SunWrite. SunWrite allows different fonts to be used for different pieces of text. The user has also replaced the underline characters from the report with solid underlines.

SunWrite allows the user to click on a component of text and then edit it. Figure 4-7 shows the various options available for a component. Different fonts, faces, and sizes are available to change the appearance of the component. The user can also specify formatting information such as a big first letter or justification of text for paragraphs.

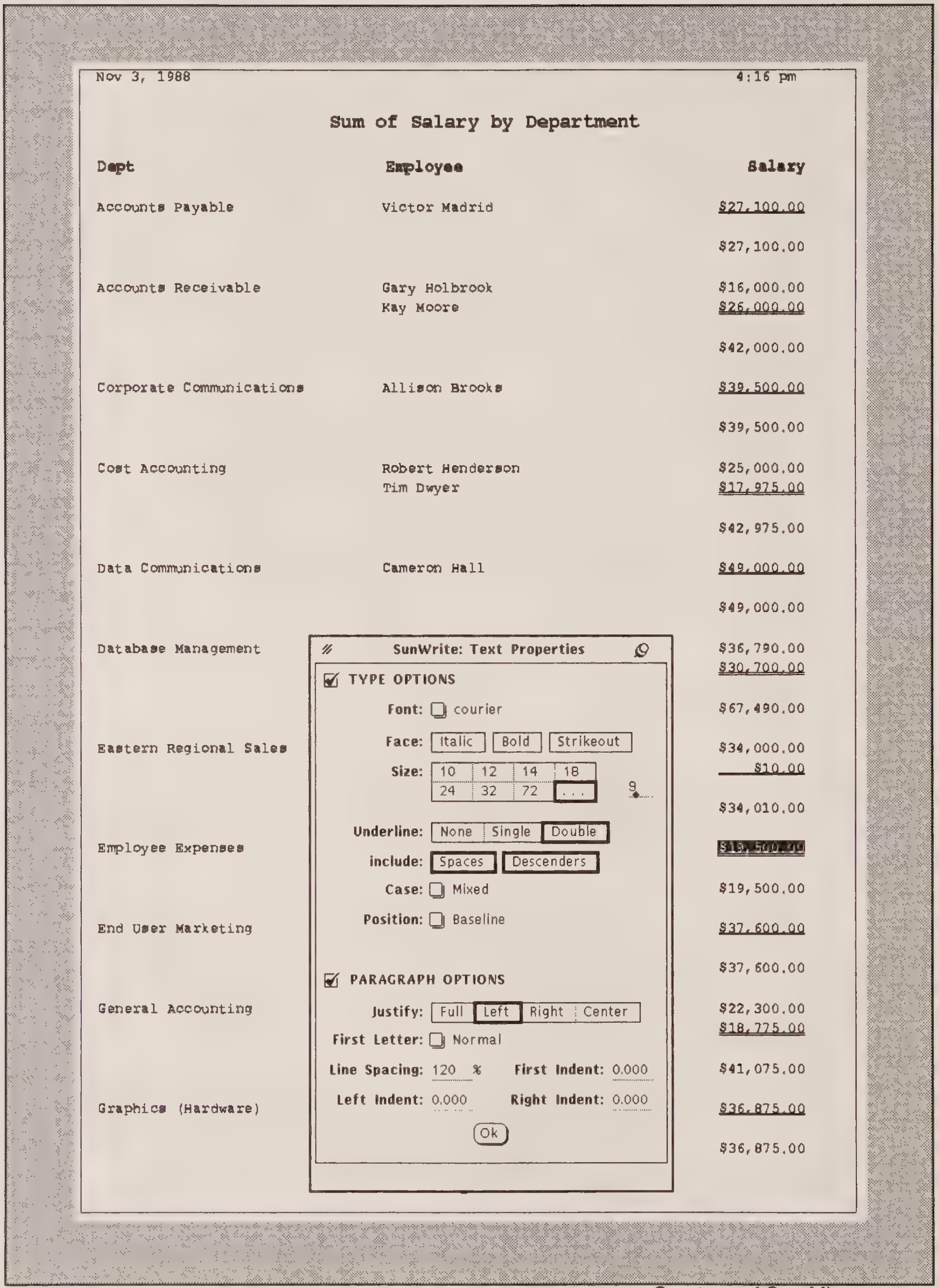
Schema Design

The schema design tool allows a user to graphically specify the structure of a database. Instead of writing SQL CREATE statements, the user can draw a series of boxes on the screen. The boxes then have arrows pointing to other boxes, specifying foreign keys for the data.

The schema design tool does not provide all of the modeling support that INGRES/-team*work* does. Team*work* is discussed in Chapter 11, on computer-aided software engineering (CASE). The Simplify schema design tool is oriented toward end users to allow them to quickly put together a database. Team*work*, by contrast, is aimed at the professional application developer.

Since both sets of tools work on INGRES databases, it is possible to use them in a complementary fashion. A prototype database can be quickly put together in Simplify, with a few reports and predefined queries. The schema for the database can then be loaded into the CASE environment for a more rigorous development effort.

Figure 4-8 shows an example of the schema design tool. On the screen is a graphical representation of the tables in the database and their relationship to each other. The lines between the tables are known as foreign keys. For example, the "nickname" table has a foreign key "real_name." The foreign key is a column in the nickname table. The foreign key allows the two tables to be joined together by matching the real_name column with the name column in the emp table.



Nov 3, 1988 4:16 pm

Sum of Salary by Department

Dept	Employee	Salary
Accounts Payable	Victor Madrid	$27,100.00
		$27,100.00
Accounts Receivable	Gary Holbrook	$16,000.00
	Kay Moore	$26,000.00
		$42,000.00
Corporate Communications	Allison Brooks	$39,500.00
		$39,500.00
Cost Accounting	Robert Henderson	$25,000.00
	Tim Dwyer	$17,975.00
		$42,975.00
Data Communications	Cameron Hall	$49,000.00
		$49,000.00
Database Management		$36,790.00
		$30,700.00
		$67,490.00
Eastern Regional Sales		$34,000.00
		$10.00
		$34,010.00
Employee Expenses		$19,500.00
		$19,500.00
End User Marketing		$37,600.00
		$37,600.00
General Accounting		$22,300.00
		$18,775.00
		$41,075.00
Graphics (Hardware)		$36,875.00
		$36,875.00

SunWrite: Text Properties

☑ TYPE OPTIONS

Font: □ courier

Face: Italic Bold Strikeout

Size: 10 12 14 18 24 32 72 [✓ ...] 9

Underline: None Single [✓ Double]

include: [✓ Spaces] [✓ Descenders]

Case: □ Mixed

Position: □ Baseline

☑ PARAGRAPH OPTIONS

Justify: Full [✓ Left] Right Center

First Letter: □ Normal

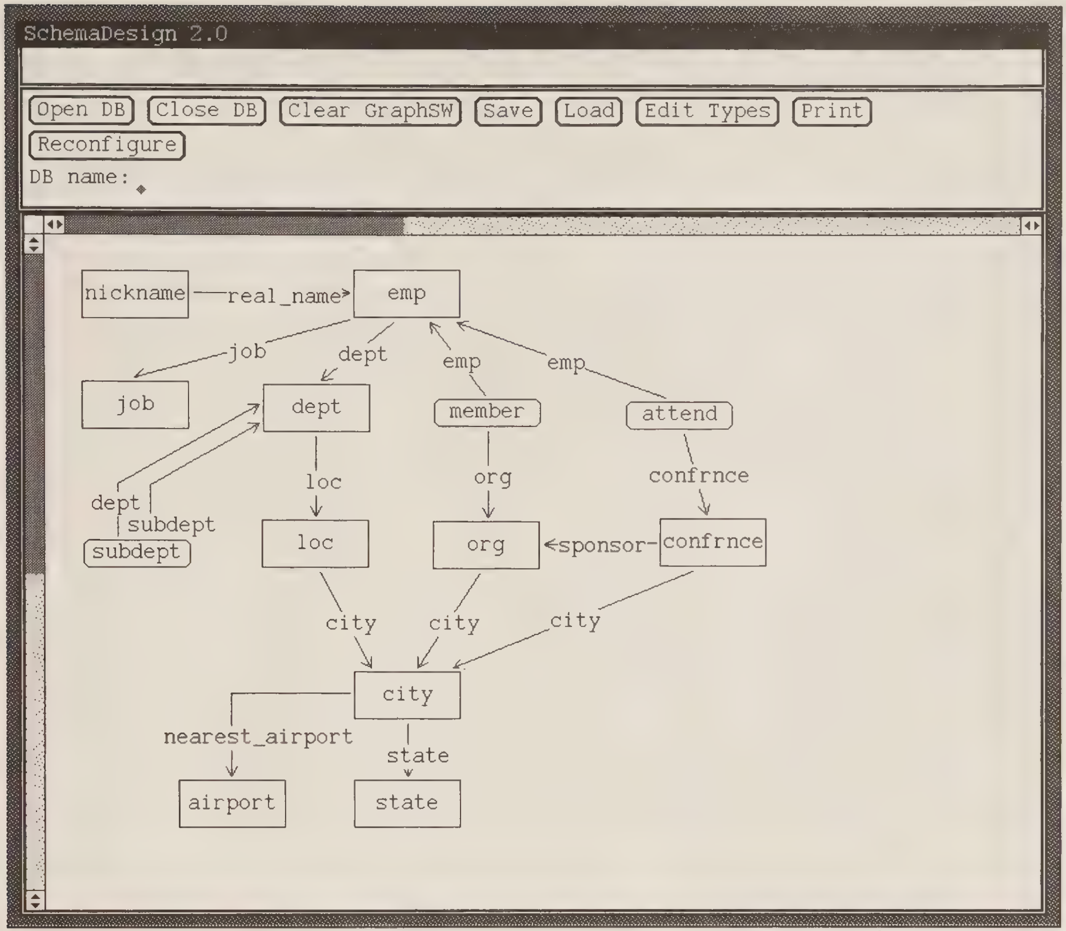
Line Spacing: 120 % First Indent: 0.000

Left Indent: 0.000 Right Indent: 0.000

Ok

Courtesy of Sun Microsystems

Fig. 4-7 Moving a Report into SunWrite



Courtesy of Sun Microsystems

Fig. 4-8 Simplify Schema Design Tool

Each of the square boxes in the diagram represents an entity, such as employee or department, which is represented as a table in the database. The rounded boxes in the schema design tool are examples of relationships. An employee is a member of an organization. The round box with *MEMBER* represents that relationship.

Relationships are also stored as tables in the database. A relationship has two foreign keys, one for each of the entities it represents. The member table, for example, has a column listing both the employee name and the organization. Employee name is a foreign key to the emp table, while organization name is a foreign key to the org table.

Interface to Command Utilities

Many INGRES database utilities are available from the operating system command prompt. For example, the user can start QBF using the following command on VMS:

```
vms$ qbf person_db -mappend -t emp
```

This command clears the terminal screen and starts up QBF using the personnel database on the emp table. The user also specified that QBF should use a table field and go directly into append mode.

Simplify allows any of the INGRES subsystems, such as QBF, as well as database administration utilities, such as the *createdb* command, to be accessed from within a window-based environment. Figure 4-9 shows the INGRES Utilities window in Simplify.

When a user clicks on the QBF option, he is presented with a submenu containing a list of available command line options and their meaning. The user can then select append mode or fill in a table name or any of the other options available. After selecting an option, a window opens to run QBF. This window is a terminal emulator—it looks to INGRES just like a single-window terminal in a more traditional environment. Of course, with Simplify, it is possible to have several such windows open, each performing a different task.

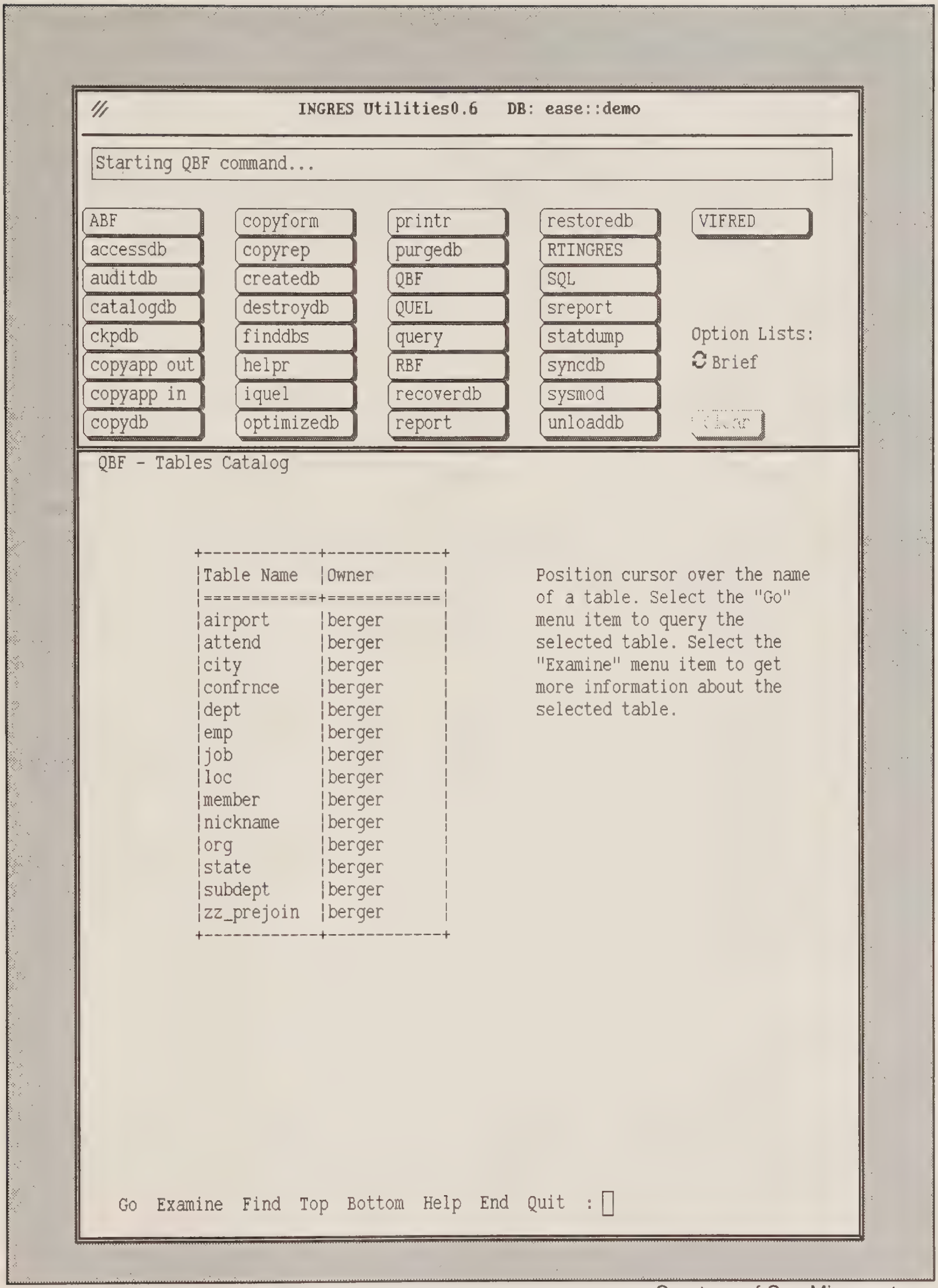
It is also possible to modify the operations of the INGRES Utilities to customize them for certain classes of users or to add new applications developed by users. Simplify reads the definitions of available options from a configuration file called *util_config* whenever the INGRES Utilities are started. Normally, the util_config file is located in a central location for all users. It is possible for a user to put a custom file on her own home directory that contains a list of all of the available menu choices that will appear in the dialogue box, along with the command associated with that option. The user can also define command options, which will appear on a submenu.

Picasso—Object-Oriented User Interfaces

The INGRES tool set has many characteristics of an object-oriented development environment. Objects, such as frames, forms, or reports, are stored in the systems catalogs. Consequently, an object is sharable and is subject to the full control of the database system. The Picasso project at the University of California at Berkeley is an example of an extension to the user interface to meet several advances in technology, including bit-mapped graphics workstations. The Picasso project is directed by Professor Lawrence A. Rowe, one of the founders of Relational Technology. In the past, many of the features to be found in the applications development environment and the user interface of INGRES have come from Professor Rowe's research projects. The INGRES forms interface and ABF, for example, are offshoots of an earlier project called FADS.

This section discusses the goals of the Picasso project. This discussion does not imply that this direction will be taken by Relational Technology. As in all major corporations, profitability, backward compatibility, and other issues alien to a research environment have a great effect on the incorporation of new technology.

Picasso is one of two important research projects at the University of California at Berkeley that are discussed in this book. Chapter 7 discusses Postgres, which is an



```
Starting QBF command...
```

QBF - Tables Catalog

```
+------------+------------+
|Table Name  |Owner       |
|============+============|
|airport     |berger      |
|attend      |berger      |
|city        |berger      |
|confrnce    |berger      |
|dept        |berger      |
|emp         |berger      |
|job         |berger      |
|loc         |berger      |
|member      |berger      |
|nickname    |berger      |
|org         |berger      |
|state       |berger      |
|subdept     |berger      |
|zz_prejoin  |berger      |
+------------+------------+
```

Position cursor over the name of a table. Select the "Go" menu item to query the selected table. Select the "Examine" menu item to get more information about the selected table.

Courtesy of Sun Microsystems

Fig. 4-9 Integration with INGRES Utilities

extension to the data manager. Picasso uses the services of Postgres to store data and to ensure the consistency of that data in a concurrent environment. Picasso also uses several enhancements to the relational model in Postgres to coordinate user interfaces that share the same objects.

The principle goals for Picasso are to provide:

- an end-user tool to develop applications with workstation interfaces
- an object-oriented programming language
- a graphical interface to edit all components of an application
- interface extensibility, such as allowing the programmer to add new types of fields to the forms system
- application generator extensibility

Workstation-based interfaces are superior to terminal interfaces because they allow more information to be displayed on the screen and allow the user to switch rapidly between tasks running in separate windows. Window-based applications are easier to learn and use because they present a more intuitive form of interaction with programs.

Object-oriented programming is a style of programming that encourages code sharing and modularity. Programs are specified as a collection of objects with attributes and procedures that manipulate the objects. The time required to develop a user interface can be substantially reduced using an object-oriented programming language and an interactive development environment that includes an extensive library of predefined interface objects.

A graphical interface allows the user to edit an object while looking at it. The current INGRES tool set provides a direct manipulation editor for forms (VIFRED) and reports (RBF) but not for frames in an application. Picasso is exploring an alternative conceptual model for frame editors. Finally, Picasso allows programmers to add new field types to the forms system to display the new data types that can be added to Postgres. Programmers can also add new application generators. An organization might create an application generator to produce a common set of frames that is used in every application, but with slightly different operations in each. This would thus be the equivalent of a custom version of ABF.

Shared Object Hierarchies

A shared object hierarchy is one of the key characteristics of the Picasso project. The shared aspect of the hierarchy means that different applications can access the same object. This feature of an object allows fairly sophisticated applications to be developed that all work together. For example, a user could be displaying a report that has several objects associated with it, including an object containing data from a database table. A user could decide to quickly graph the data, and since the data are objects, they can be shared by the graph utility instead of having to be regenerated by the application.

Sharing of objects is done through the services of the Postgres data manager. Postgres has a feature called a trigger or alerter that allows an application to be notified whenever a particular event occurs on database data. In this case, the front-end application maintains a cache of objects in it's own memory space. Whenever another application changes the object, the first application is notified by a trigger. The application then goes back to the database to get the newest version of the data.

Sharing objects by storing them in the database allows the use of all of the services of a database manager, such as query optimization and locking support. Programmers are thus freed from developing code to ensure the consistency of objects across multiple applications or displays.

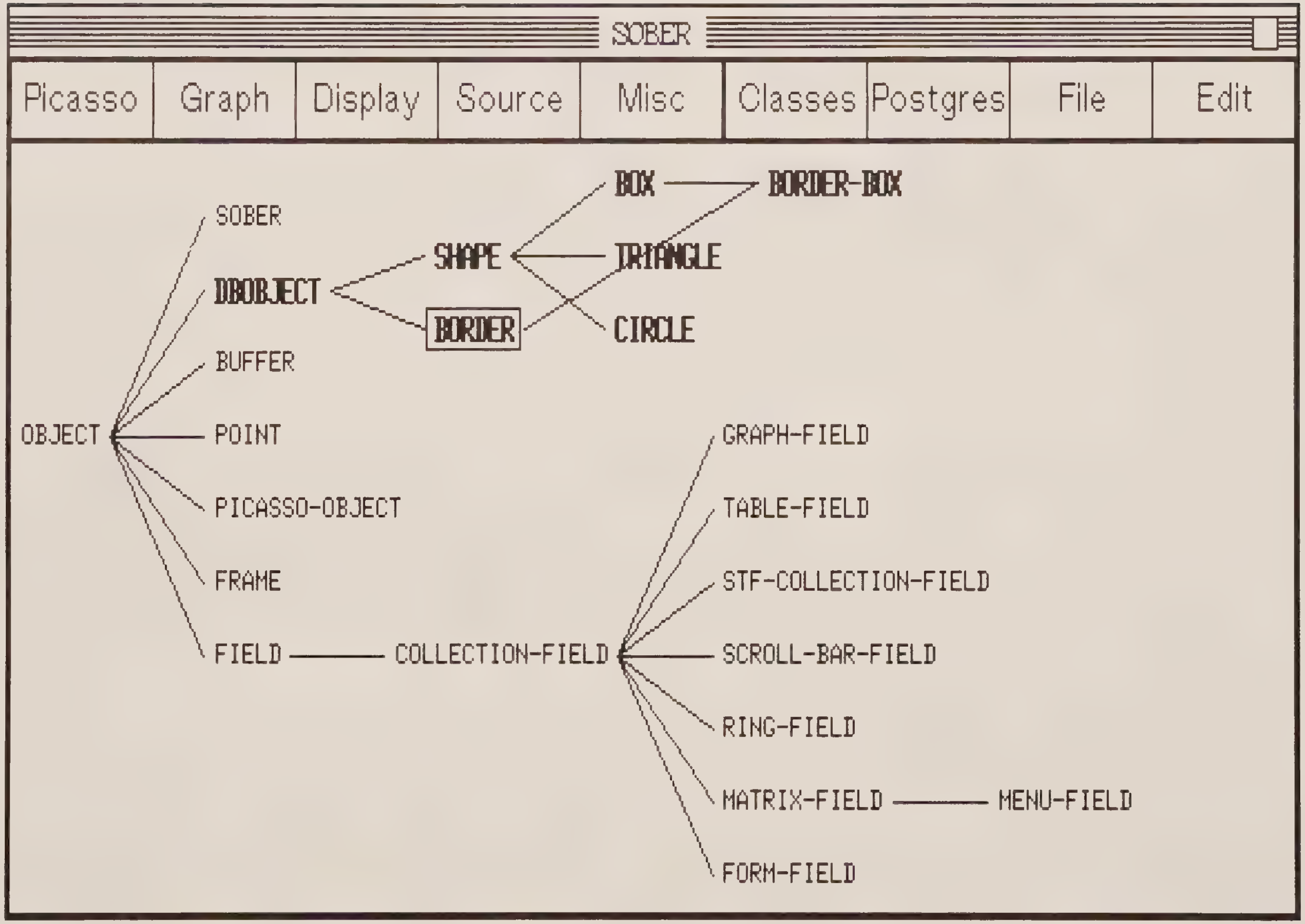
Keeping all objects in the database is fine for concurrency, but does not necessarily help performance. For an application to run effectively, it needs to maintain the objects in main memory. This main memory cache is private to the application and allows it to quickly access an object, such as the next form to display. Picasso balances the need to protect objects in the database with the need for high performance by individual applications. The other alternative is to not use the services of a database manager. This means that each programmer has to reinvent the services of a database, such as how to store and retrieve complex objects.

The hierarchy aspect of the shared object hierarchy also has important advantages. Each object has a set of characteristics associated with it, such as the format of the object, and operations defined on the object, known as methods. By structuring objects in a hierarchy, a low-level object does not have to duplicate the functions, that is, the code, already in the higher-level objects. Take, for example, the object known as a field. There are a wide variety of different types of fields that are supported in the Picasso forms system. The generic object "field" has a series of methods that can be used by all of the different types of fields. Lower in the hierarchy are specialized fields, each with their own methods and characteristics associated with them.

A graphical editor is part of the Picasso development environment and allows the user to quickly examine and modify the object hierarchy. A new type of object, for example, can be inserted at a particular point in the tree. In addition, the user can examine the methods defined on objects. Figure 4-10 shows an example of the graphical object editor. All of the objects on this hierarchy are displayed in a window. Each of the objects in the hierarchy corresponds to a similar object in the database. A procedure is used to translate the object from database format into the format needed by the front end system, such as a common Lisp program.

Picasso uses a feature of Postgres known as inheritance. Inheritance allows database tables to be stored in a hierarchy. A lower-level table inherits all of the attributes of a higher-level table. This conforms nicely to the shared object hierarchy used by Picasso in the front-end systems. In Figure 4-10, the hierarchy includes fields, frames, and other aspects of a forms-based user interface. The object field includes a variety of subobjects, such as a graph field or a table field.

Another form of object in the hierarchy is the database object, corresponding to data in the database. These objects include a variety of graphical data types such as boxes, triangles, and circles. The user has added a new type of object called a border. The object *BORDER-BOX* inherits methods and attributes from *BOX*, *SHAPE*, and *BORDER*, in addition to having its own methods and attributes.



Courtesy of University of California

Fig. 4-10   An Object Hierarchy Graphics Browser

Complex Objects

A table field in INGRES is an example of a complex object. The table field is an object in its own right and certain methods are attached to it. For example, a method associated with a table field is the initialize operation, which sets up a portion in memory to hold the data set associated with the table field.

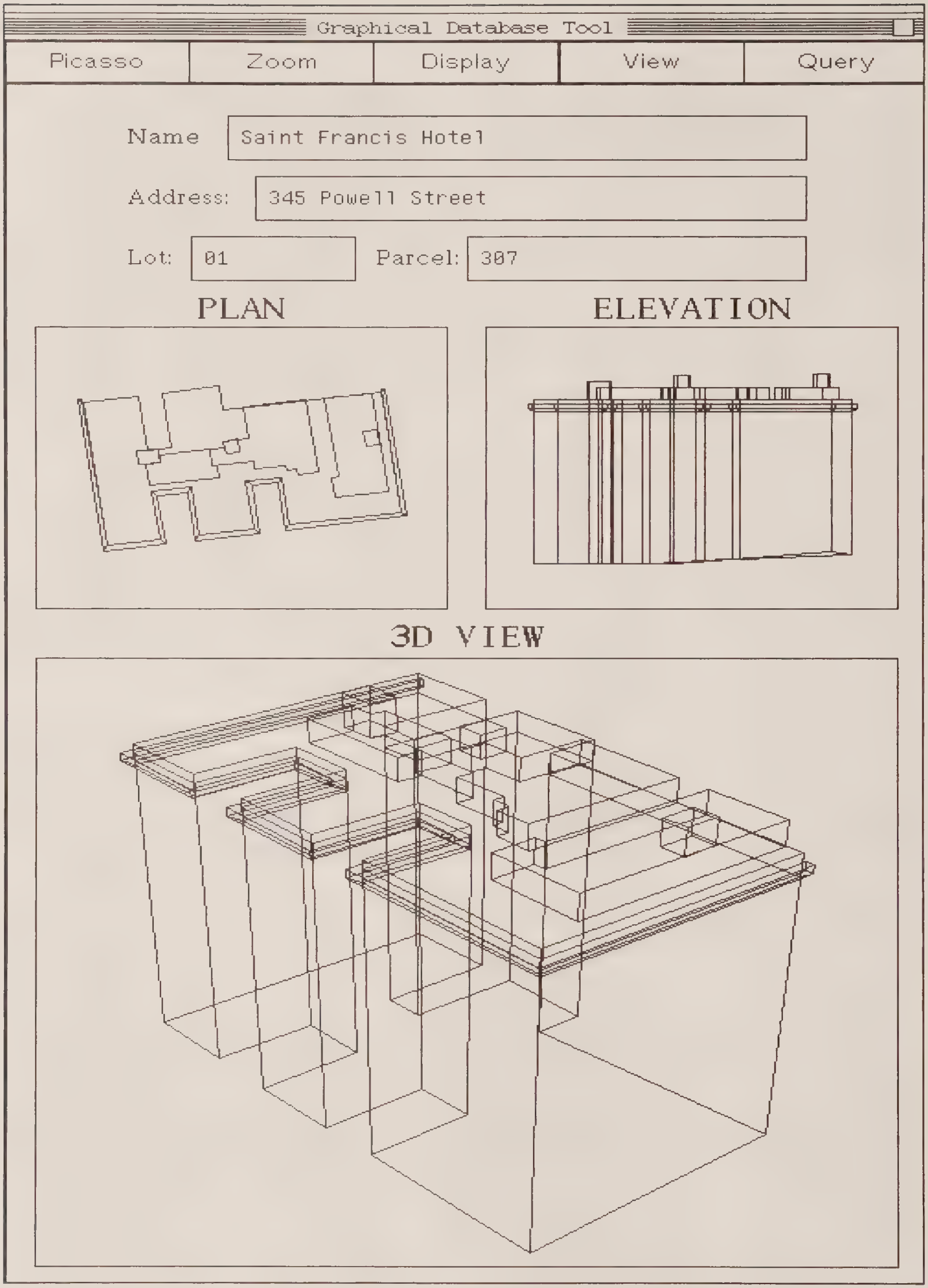
Other methods associated with a table field are a variety of user interface operations. When the user hits the down arrow, this activates a little procedure that moves the cursor down to the next row of the table field. If the cursor is on the last row of the table field, the data set is scrolled up to make the next row visible. Finally, if the last row of the data set is displayed, the "Out of data" message is displayed on the screen. Within a table field are a series of fields, corresponding to the intersection of a row and a column. These fields have all the attributes of a simple field. Data can be displayed and validation checks activated, for example.

One of the goals of Picasso is to allow for a wider range of complex objects. The fields of a table field could thus be a variety of different objects. One field could display a graphic image; a second could be a simple textual display. Another example of a complex object would be a spreadsheet table field. The cells of the spreadsheet are represented by fields. Different cells are related to other cells. These cells could thus be a formula based on a region of the table field. When data are displayed in the spreadsheet-like table field, the calculated and derived cells are all adjusted based on the data in the cells containing the base data from the database.

Because the underlying database used by Picasso, Postgres, is extensible, it will also be possible for users to define their own objects. A field on the form, either a simple field or a cell in a table field, could be defined by the user to have certain display characteristics. Procedures would then be defined that instruct Picasso how to get the data out of the underlying database and translate it into the appropriate form of display.

An example of a new form of object would be voice. Digitized voice could be stored in the database. The display characteristic for digitized voice could be a blinking star. Associated with this object would be a method called "Play_Voice." When the user clicks the mouse on this field, the Play_Voice procedure would take the digitized voice and output it to the speaker on the workstation.

Figure 4-11 shows a prototype of an application that puts a variety of different fields on the screen. This application shows the floor plans for different buildings in San Francisco. Each of the different graphical views of the data is derived from the same tables in the database. The code associated with the *PLAN* field extracts the floor plan from the database while the *3D VIEW* field shows a three-dimensional view of the building.



Name: Saint Francis Hotel

Address: 345 Powell Street

Lot: 01 Parcel: 307

Courtesy of University of California

Fig. 4-11 A Complex Form in Picasso

Summary

Bit-mapped workstations and powerful personal computers have led to a variety of different styles of user interfaces. The Apple Macintosh, for example, has a certain look and feel. A look and feel means that different programs tend to all interact with the user in the same fashion. A help key might always be a certain key on the keyboard, or selecting a menu option is always the same.

INGRES operates in a variety of different workstation environments. The X Windows System developed at MIT and incorporated in products from DEC is one environment. The Apple Macintosh is a second. IBM's Presentation Manager, part of the System Applications Architecture, is a third. Finally, the Open Look environment that runs on AT&T, Xerox, and Sun workstations is yet another.

One of the challenges for Relational Technology is keeping INGRES portable in this environment that is characterized by different user interface standards. In the terminal environment, INGRES uses a common set of tools that are portable across different operating systems. INGRES is able to adapt to different types of terminals by defining logical functions in the INGRES 4GL. At run time, these logical functions are translated into specific operating system calls and were mapped to specific keys on the terminal. As users move from character terminals to bit-mapped workstations, a variety of efforts are underway to provide tools that take advantage of these more powerful environments. These new user interfaces take advantage of the graphical power of workstations to display data in the form of icons, windows, and other idioms.

Simplify is one example of a commercial system that allows workstations to be used with INGRES databases. Simplify has a set of tools that allow the user to construct queries, run reports, and even construct new databases.

The Picasso project is a longer-term effort trying to construct new forms of user interfaces that take advantage not only of the workstation user interface but of new programming techniques. Picasso uses the services of the Postgres data manager to allow shared object hierarchies and complex forms to be incorporated into applications.

The three chapters in this part of the book have presented a wide variety of different user interfaces, ranging from simple SQL to QBF to bit-mapped workstations. All of these interfaces share a common characteristic—they generate SQL statements and dispatch them to the data manager.

The next two parts of this book examine what happens after the SQL statements are generated. Part II looks at how the data manager takes an SQL statement and turns it into low-level requests for data on a computer while still maintaining high performance and the integrity of the data. Part III then examines how the SQL is transmitted from the front end to the back end over a network. Part III also shows how these user interfaces, such as QBF, are able to access non-INGRES data repositories through gateways.

Part II

The Data Manager

Overview

Part I of this book concentrated on the user interfaces for retrieving and displaying information. In discussing the construction and use of a user interface, we assumed the presence of a data manager that could respond to standard SQL (or QUEL) requests for data. In this section, we look at the other side of the picture: how the data manager processes a query.

In the previous part, the data manager was a black box. For this part, the user interface is a black box—it could be QBF or a complicated Embedded 4GL program. Both programs operate the same way from the standpoint of the data manager: SQL statements are accepted and error messages or data are returned.

Chapter 5 discusses how a manager efficiently retrieves information for a user. The query optimizer is responsible for choosing among a variety of different ways of getting data, called access plans. The query optimizer decides on the optimal method for getting the data from the files in the database and formulates a query execution plan.

Chapter 6 adds a further level of complexity by showing how INGRES is able to provide access to many different users, each with her own query execution plan. Chapter 6 also shows some of the methods INGRES uses to make effective use of the configuration of a computer.

Chapter 7 discusses some extensions to the data manager being implemented as part of a research project at the University of California at Berkeley. This project, Postgres, gives the reader a glimpse of what a database system might look like in a few years.

Chapter 5

Efficient Data Retrieval

This chapter focuses on the efficient retrieval of data in a single user environment. When an SQL query is received, the request for information is in a logical form—users ask for data in terms of tables and columns. The data manager must translate this logical request for data into low-level requests for records from the underlying files that make up the database.

In order to access data efficiently, INGRES supports a variety of different storage structures for database tables. A storage structure provides an index to the data in a table based on the values in one or more columns, called the primary key for that table. The index allows a particular record to be directly accessed using a key instead of searching all records in the table.

There are often times when users wish to access data, but do not know the value of the primary key. Instead, users specify a search based on the value of the data in some other column in the table. Secondary indices allow the primary storage structure of a table to be supplemented by alternative direct-access methods. In a table with a primary storage structure and a number of secondary indices, there are a variety of ways to satisfy any given request for data. The query optimizer examines all the possible access plans to decide which one is appropriate for a given query.

In addition to efficient access to data, the back end provides a variety of other services. Among them are the ability to provide security constraints on access to data to prevent unauthorized access. A related service is the integrity constraint that prevents data from being added to tables that violate the constraint. Both types of constraints, by being built into the back end, apply to all types of front-end tools.

In both this and the next chapter we assume that both front and back ends are located on a single computer. As will be seen in Part III, it is possible that the data managers are distributed throughout a network. A front end is able to use the services of the General Communications Facility (GCF) to access any back end throughout the network. Since this chapter is concerned with the operation of the data manager, we assume for the time being that a front end is somehow able to communicate its requests to the data manager and ignore the details of how that communication takes place.

Query Processing

When a query is received from a front-end, the data manager goes through several steps to process that query (see Fig. 5-1). First, the query is scanned, validated, and parsed. Scanning identifies the parts of the query. Object validation ensures that all tables and columns referenced exist and that data types referenced in the query are compatible. Parsing ensures that the query is written correctly. If there is an error, such as an SQL syntax error, an error message is returned to the application.

Next, the query is modified to include permits and integrities, a series of rules stored in the database that modify the effect of a query. For example, a permit is a rule that grants users the ability to access (or not access) certain tables. Whereas a permit is based on users, integrities are based on the underlying data. An integrity can prevent bad data from being entered into the DBMS. An integrity thus serves many of the same purposes that validation checks do in VIFRED. The difference is that if a user doesn't use a VIFRED form, the validation check is bypassed. With an integrity definition, the validation is always performed.

Along with modifying the query to include integrities and permits, any views referenced in the query are expanded to their underlying definition. Views are virtual tables: the user thinks he is seeing an actual table in the database. In reality, the view defines an SQL select statement that is executed every time the view is accessed. Views help simplify the appearance of the underlying database by providing customized tables for particular types of users.

Once the actual query is formulated, it is sent to the optimizer. The optimizer decides in which order to access different tables, and which access paths to use. A good query optimizer is the heart of a relational database system. If two different tables are being accessed in a query and each table has 10 million records, the query optimizer decides the most efficient way to access the two tables.

A bad query optimizer (or a poor database design) could result in the two tables of 10 million rows being combined to form a table of 100 trillion rows, known as the cartesian product of the two tables. Alternatively, the query optimizer could use various indices to perform only a few hundred accesses to the table. Needless to say, this makes quite a difference in performance!

Once the query optimizer has formulated a query execution plan, the query execution facility reads the plan and sends a variety of requests to the Data Manipulation Facility (DMF). It is the responsibility of the DMF to perform low-level access operations to the files that make up individual tables. Blocks of data are read from the file, filtered, and sent back up to the query executor, which processes the data according to the requirements of the query execution plan.

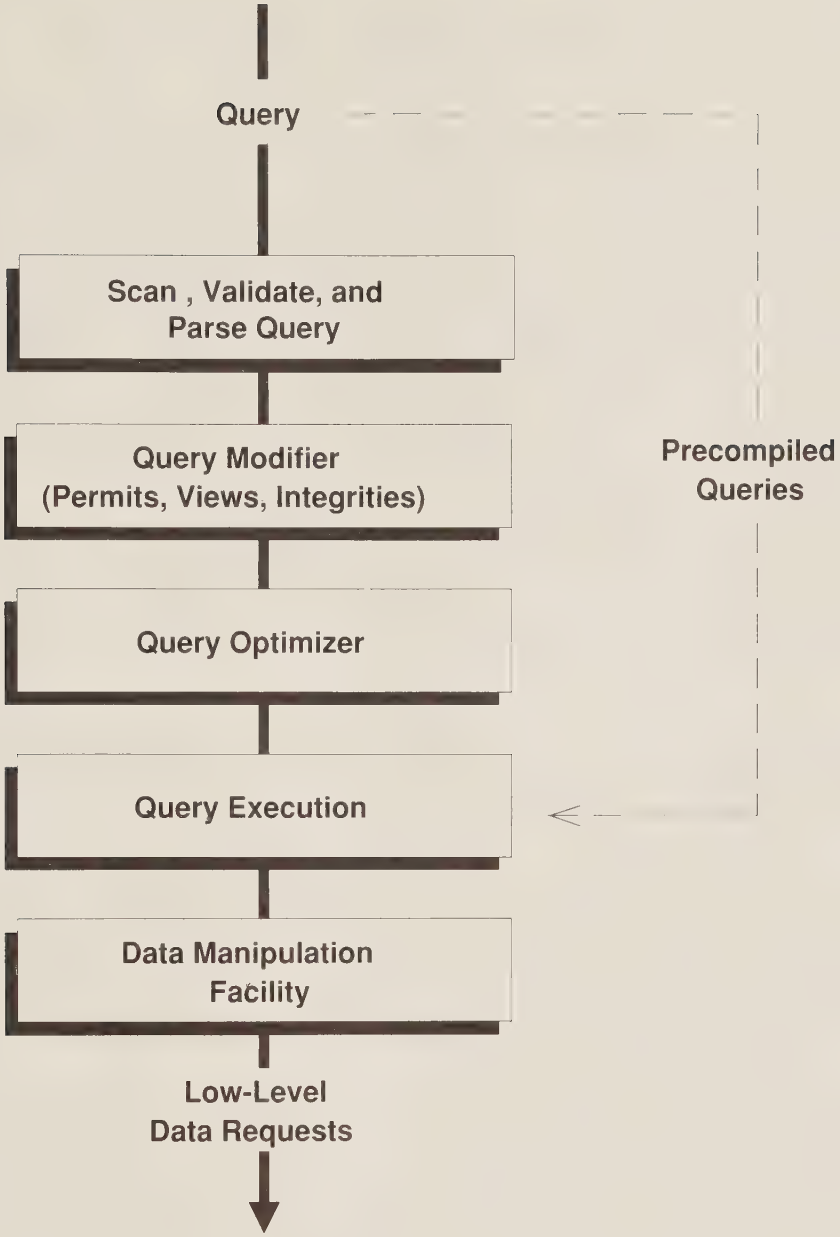

Query
Scan , Validate, and Parse Query
Query Modifier (Permits, Views, Integrities)
Query Optimizer
Query Execution
Data Manipulation Facility
Low-Level Data Requests
Precompiled Queries

Fig. 5-1 Steps in Query Processing

We will discuss these three levels of processing a query in reverse order. First, efficient access to a low-level file is discussed. This is the question of the proper storage structure and the use of secondary indices. Next, the query optimization process is discussed. The query optimizer uses the facilities of DMF, but decides in which order data are to be accessed. Finally, we will discuss views, permits, and integrities—three methods for modifying a query without the knowledge of the user.

Storage Structures and the Data Manipulation Facility

A storage structure determines how the data in a particular relation are stored on disk. The users of SQL do not need to concern themselves with this issue; no matter what the storage structure, data are always accessed by using an SQL statement. This means that storage structures can be designed in a very simple way in the beginning stages of the information system. Later, the storage structures can become more complex, but none of the applications need concern themselves with these issues. The separation of the logical organization of the data from the physical implementation is one of the strongest advantages of the relational database environment.

The actual retrieval of data is done by the DMF part of the data server, which handles many of the tasks that programmers of a nondatabase application would have to do, including opening a file and requesting certain pieces of data to be read or written. Each table in INGRES is stored in a separate file. These files are broken up into blocks of information known as pages. Each page holds 2048 bytes of data. The page may contain data, may have empty or unused space, or may contain indices to data. The actual composition of pages in a file varies, depending on the storage structure of that file.

Each database row stored in a page has an identifier attached to the data called the Tuple Identifier (TID). The TID consists of two pieces of information: the page number of the data containing the row and the offset of the row within that data page.

When DMF retrieves data, it does so in units of pages, which may contain rows. This means that a request for a single row of data may actually read several. One of the functions of the data server is to cache data pages in main memory. Often, the user needs a subsequent row, as in the case of a query that retrieves all employees with an age greater than 20 years. If so, since the pages are cached, there is a good chance that the next row needed is already in main memory, alleviating the need to perform a much slower access to the disk containing the file.

A storage structure establishes how data are arranged in the pages of the file that make up the table. Some storage structures allow keyed access to data, meaning that the query executor can request that the DMF return all pages where the key has a certain value. Keyed access to data means that the query executor does not have to request every page in the table in order to determine if there are any rows meeting the search criteria. If keyed access to the data is not suitable, then the query executor has to request all of the pages, known as a scan of the table. As will be seen, different storage structures direct access to data for different types of queries. There are four storage structures that INGRES can use for files:

- heap
- ISAM
- BTREE
- hashed

These structures will be discussed in turn.

Heaps

A heap is the default storage structure for INGRES tables. The heap consists of an unordered set of rows. When a new row of data is added, it is put at the end of the file. The heap structure thus provides no direct access to data. When data are retrieved, the data manager must scan every page in the heap to determine if any rows meet the search criteria. This is equivalent to a desk with a bunch of papers stacked up on top of it.

A heap storage structure is the most efficient storage method in INGRES from the point of view of storage space, since no indices are contained in the file and no space is reserved in the data pages for future additions. When data are deleted, the space used by the deleted row is not reclaimed—all new records go to the last page in the file.

If there are a substantial number of deletions to a file, it is possible to reclaim the freed space using the *modify* command. The modify command is used to change the storage structure of a table. In this case, the command is being used to change the structure from the existing heap structure with unclaimed space to a new heap structure:

```
modify emp to heap
```

The effect of the command is to move each row in the emp table to a new version of the table. The rows are moved so that each page in the new table is filled as much as possible, thereby reclaiming space used by the previously deleted records.

Because data always go to the end of the file, adding new data to a heap is very efficient. For an update operation, however, the data manager must first find the proper record and then update it. This involves a scan of the entire table because there could be several rows that meet the update criteria. Retrievals, like updates, also require a scan of the entire table.

A special form of the heap storage structure is the heap sort. The data in the table are first sorted based on the value of one or more columns, and then placed into a heap. Sorting data before putting them into a heap can speed up the retrieval of data. When the query executor is comparing the data retrieved to some value in the search criteria, it usually needs to sort the data. Since the data in the heap are sorted, the amount of time required to sort the data is reduced (assuming the column specified in the query is the same one that the data are sorted on).

Putting data into the proper location so that I/O operations are reduced is known as data clustering. Clustering of data does not alleviate the need to scan all the pages in a heap but does reduce the number of I/O operations needed to sort the data. When the table is first modified to a heap sort storage structure, all the data are in the proper order based on the column or columns they are sorted on. As data are added to the table, they are added to the end of the heap, not into the proper location for sorted data. A heap sort thus starts out as a sorted table, but gradually degrades as updates and inserts are performed. If the underlying data change frequently, the user can remodify the table to heap sort as needed. The table is once again sorted and the data clustered. Remodifying a table also reclaims any free space left over from previous delete operations.

The heap storage structure thus has the advantage of being efficient from the point of view of storage space and adding new rows to the table. It has the disadvantage of not being very efficient for retrieval of data. For direct access to data based on key columns, other storage structures (or secondary indices) are used.

ISAM

The indexed sequential access method (ISAM) is an alternative to storing data in a heap storage structure. With an ISAM table, the data are indexed, and, when data are retrieved, updated, deleted, or inserted the index can be consulted to find the proper location for the data. The index on an ISAM table is based on a particular column or set of columns in the table, known as the primary key. For example, in an employee table, the primary key might be the employee name. If the primary key is the employee name, and a user requests a particular employee, the index would be consulted, which would then point to the particular page in the file that contains the relevant row or rows.

If the index is the employee name, however, lookups based on the social security number or other columns of the table would require a scan of the entire table. This is because these other columns are not part of the primary key. For example, if the user requested a row where the social security number had a specific value, the DBMS would have to scan all pages in the table and return all rows that matched the social security number in the query.

The choice of what columns of a table to put in the primary key is an important one. When the query optimizer examines a query, it looks for access paths to the table based on the value of a key. For example, the following two queries access the same table, which has a primary key based on the name column:

```
select * from emp where emp.name = "Jordan"
select * from emp where emp.salary > 2000
```

In the first query, the optimizer knows that it will be able to take advantage of the ISAM structure of the table. It thus tells the DMF to access the base table using the primary key. For the second query, it knows that the ISAM table is not indexed on salary, and thus tells DMF to scan all the pages and return those rows with a salary of greater than 2000.

Tables are initially formed as heap storage structures. If the database administrator knows that access to the employee table is often done on employee name, the following command can be issued:

```
modify emp to ISAM on name
```

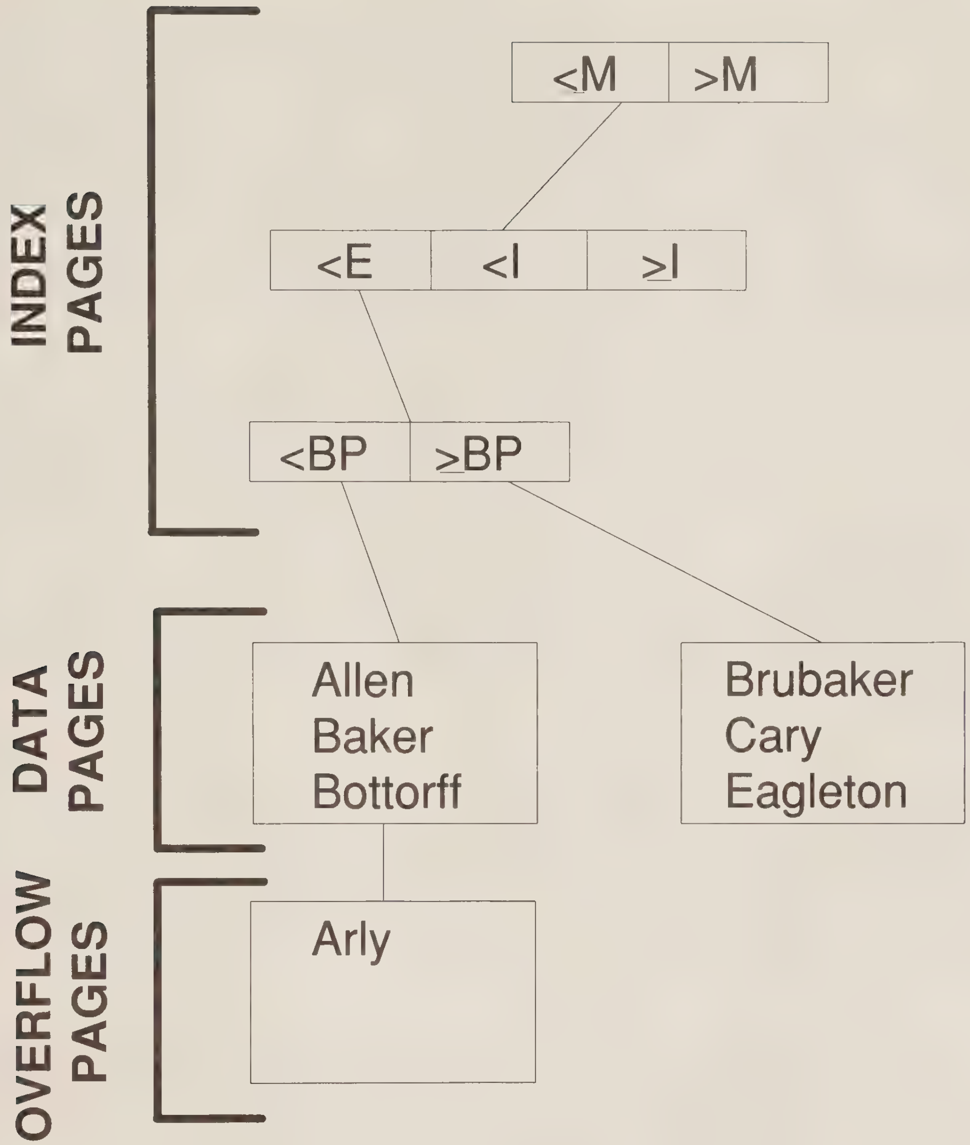


Fig. 5-2 ISAM Storage Structure

The effect of this command is as follows. First, all the data are sorted based on the key values and placed into pages of data. The data are now clustered—a page of data will have several rows of data in it, all with similar key values.

Next, an index is constructed on the underlying data. This index is composed of several levels. The top level of the index points to the next level, which points to a lower level, which eventually points to the actual data pages that contain the rows of data with the proper key values. Figure 5-2 shows the different portions of the ISAM table.

The top level of the index is an index to the next level. In the employee name example, the top level might point to one part of the second level for all names beginning with A to M and another part of the second level for all names beginning with N to Z. The second level is again an index to the next level. The A to M part of the second level would point to various parts of the third level, depending on the value of the indexed column. At the lowest level of the index, a more precise value of the key is contained, which points to a data page.

Because the ISAM index is built at the time that the storage structure is first created, it is known as a static index. If new rows of data are added, INGRES attempts to put them onto the proper page of data based on the index values. If the pages pointed to by the index are full, INGRES creates overflow pages. The data manager would go to the index, which would in turn point to data pages. The data pages in turn have pointers to the overflow pages.

As will be seen, a large number of overflow pages can lead to bad decisions by the query optimizer and also lead to concurrency problems. The solution to a large number of overflow pages is to remodify the table. The modify operation will re-sort all of the data, and then reconstruct the index.

One solution to reducing overflow pages is to leave extra space in the data pages when the ISAM structure is created. When executing the modify command, the user can specify a “fill factor.” A fill factor indicates that only a certain percentage of each data page should be filled with data, leaving room for new additions to the table.

The depth of an index, the number of levels, influences the number of I/O operations needed to access a single row of data. If there is a three-level index, the data manager must first access three pages of index, then the page of data, resulting in four I/O operations.

An important feature of the INGRES data manager is that indices to frequently accessed tables can be cached in the main memory of the computer. Instead of having to actually access the underlying file, the index can be consulted at the comparatively much greater speeds of main memory. Although CPU cycles are still required to process the index, I/O operations are skipped when the index is cached.

The ISAM storage structure allows direct access to data based on key values. ISAM allows the user to specify a partial key value and still take advantage of the storage structure. The index is constructed by sorting the keyed columns. If a query had requested all names with a where clause *name = "M%"*, the index can be consulted to find the region of the index containing those values. The index would be used to narrow the search down to those data pages containing rows starting with the letter M.

On the other hand, if the user formulates a query that doesn’t specify the left-hand part of the key, the whole table has to be scanned. For example, the following query requests all names with an M somewhere in the name:

```
select * from emp where emp.name = "%M%"
```

In this case, the index does no good because the index was constructed by sorting the name column with the first letter being the most significant. There is no index based on the middle letters of a name. ISAM is thus very effective for partial queries if the query specifies the left-hand part of the key.

It is possible to specify several columns of a table as the key for an ISAM storage structure. This is known as a concatenated key. In the employee example, we could make the combination of first name and last name the primary key of the table:

```
modify emp to isam on last_name, first_name
```

The data are sorted first by last name and then by first name.

If the key is constructed on last name and first name, a query asking for *last name = "M%"* would still be effective and would make use of the ISAM structure because the query specified the left-hand part of the key. However a query that asked for all first names beginning with C would not make use of the ISAM index because the data were sorted first on last name. A first name beginning with C could be anywhere in the table, so the table is scanned.

ISAM is a storage structure consisting of an index to data pages based on the value of a key column or columns. The storage structure is static, so any rows added after the index is formed that do not fit into the data pages are placed into overflow pages. ISAM is able to provide direct access to data based on the key columns if the left-hand portion of the key is specified in the query.

BTREEs

A BTREE (binary tree) storage structure also uses an index to access data. It has two important differences from the ISAM structure. First, the lowest level of the index, known as the leaf level, contains a pointer to each row of data. The second difference is that the index is dynamic. When new data are added the index structure is updated if necessary. The top layers of the BTREE index look very similar to the ISAM table (see Fig. 5-3). The top level contains a set of key values, and the pointer to the next level of index pages is based on the value of the key.

The last layer of the index points to the leaf pages. The leaf page contains the key value for each row in the table, along with the tuple ID (TID) of that row. The TID gives the page number of the row and the offset of the row on that page. The leaf level index allows the data manager to then directly access the row needed instead of searching within the data page (and the overflow pages) for the proper rows. When data are accessed via a BTREE index, they are returned to the query executor in sorted order (ISAM may not due to overflow pages).

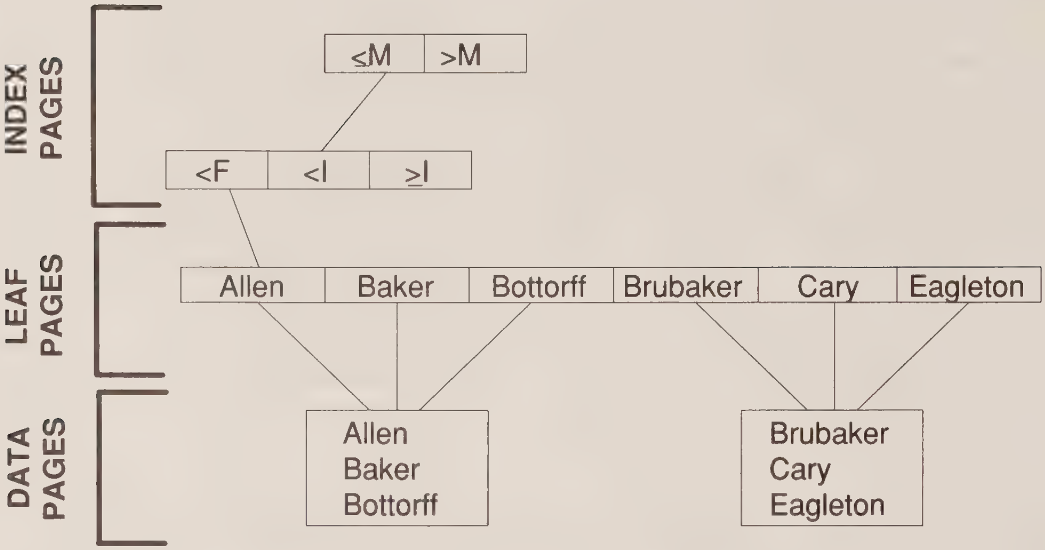


Fig. 5-3 BTREE Storage Structure

The dynamic nature of the BTREE index means that there are no overflow pages in the table. For an application where key values are being continually updated, the BTREE provides an important advantage over ISAM. However, if the data are fairly static, the ISAM structure can sometimes yield better results because a leaf page does not have to be consulted for each row accessed. Note, however, that the leaf and index pages of the BTREE might be cached, thus increasing the speed of the operation.

The dynamic index does have an important implication for concurrency. In ISAM, the index is static and thus never has to be locked when data are being changed. In a BTREE update operation, if the key value is changed, the index might have to be restructured. This can result in locking several of the index pages. Any of the index pages that are locked are unavailable to other users. In the case of a leaf page, for example, pointers to several rows of data might be on one leaf page. The rows pointed to by the locked leaf page are unavailable to other users.

Both ISAM and BTREE tables are good for range searches based on a key value, including partial key searches. The basic difference between the two is the dynamic nature of the index. If the data are static, ISAM produces higher performance. If the data change often, the dynamic nature of the BTREE index can yield higher performance.

BTREEs are also important in environments with very large tables. When a table is modified, there needs to be enough free disk space to hold both the original table, the new table, and some sort space. If there is not enough space to remodify the table, a BTREE structure may be appropriate. A BTREE can be established when the table is created, whereas the ISAM table needs to be modified after the data are added, or all data may end up in overflow pages.

Hash

A fourth storage structure is the hash table. Like BTREE and ISAM tables, there is a key defined for the storage structure. The hash structure, however, does not use an index to access data. Instead, the key value is put through a mathematical formula that yields the page address for the data.

This mathematical formula, called a hashing algorithm, transforms a key value into a page address. Because the hash algorithm works on the key value, it must have the whole key value to perform the transformation. Hashed tables are usable for partial key searches (i.e., *name = "M%"*) because the formula needs the whole key to be work.

The hashing algorithm constructed for a table is a function of the amount of data in the table and the desired fill factor for that table. The fill factor allows a page to be only partially filled with data, leaving room for new insertions. Once a page gets full, overflow pages are used, as in the case of ISAM (see Fig. 5-4).

For a query where the complete value of the key is always specified, a hashed table is more effective than an indexed one because an index does not need to be consulted. For a search based on a range of key values, however, the entire table must be scanned. This is because data that are in sorted order are not necessarily placed on the same data page by the hashing algorithm. For example, if a personnel table is usually accessed based on an individual's social security number, a hash algorithm is very effective. If access is usually on various combinations of last name (i.e., *"M%"*), each of these queries requires a scan of the table.

Hash structures are particularly good for tables with very wide or multicolumn keys. In both cases, the hashing function works equally well despite the width of the key. In BTREE and ISAM, by contrast, wide and multicolumn keys result in noticeably larger indices.

Secondary Indices and Key Design

Deciding what storage structure to use and what columns should be keys is often a difficult choice. For an employee table, access might vary depending on the application. Some users would access exact names. Others would do searches on last name. Still other users might access data based on salary. A storage structure based on one key value can only satisfy one of these users adequately.

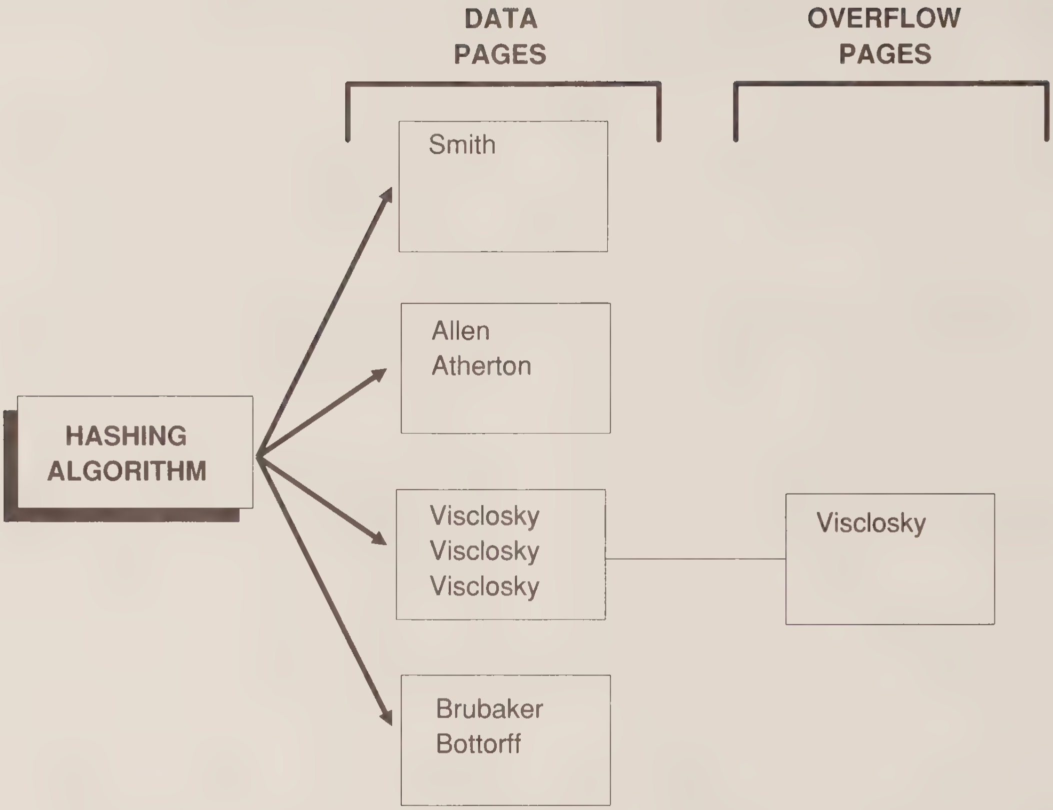


Fig. 5-4 HASH Storage Structure

When a table is modified on a particular column to a storage structure, that column is known as the primary key. The term primary key denotes that the data in the table are ordered by that key value. Deciding on the primary key and the storage structure is a matter of compromise. The database designer should determine what types of access will occur on the table, how often each type of access will occur, and the relative importance of the different types of access. In a multiuser environment, no one choice may be optimal.

To aid in this situation, INGRES allows secondary indices to be constructed on tables. A secondary index is a separate table that contains two parts: a key value and an address for the corresponding row of data in the base table. The address for the row in the base table is called tuple ID (TID) pointer. The pointer contains the page number and offset of the row within that page.

Since an index has a pointer to the TID of the base table, it is dependent on the storage structure of the base table. If the table is remodified, the rows may move, and the TID pointer in the secondary index would no longer be valid. Whenever a base table is modified, the indices are all destroyed and must be reconstructed.

The secondary index, since it is also an INGRES table, can have its own storage structure. In the employee table example, the base table might be a hash table on employee name to satisfy the common requirement of exact access based on last name. For range searches, a secondary index on the name column using an ISAM storage structure can be constructed. For salary searches, another ISAM secondary index can be defined using the salary column as a key.

There are many different ways to access data in the base table. As always, a scan of the entire table is possible. There are also several indices, both primary and secondary, available. With the secondary indices, it is possible to use the index to access the base table. Sometimes, the data manager can use data directly out of the secondary indexes, bypassing the base table.

Choosing which of these access methods is appropriate for a particular query is the job of the query optimizer. As can be seen, there are a large variety of different ways of accessing data from just one table. When multiple tables are involved in a query, the number of possible query execution plans can grow very large. Secondary indices help speed up retrieval time by allowing the query optimizer to choose among different access methods. A secondary index, however, has to be updated whenever the location of a page in the base table changes. Secondary indices can thus increase the amount of time it takes to update data in the database.

The choice of what columns to make a key, either primary or secondary, may be a difficult one for the database designer. Since users ask for data using SQL statements, they are not aware of the presence of either primary or secondary indices or storage structures. Users ask for data and get it no matter what the physical organization of the database. It is thus possible for the database administrator to experiment (prototype) various physical designs without the need for applications to be rewritten.

Three types of keys are particularly difficult to deal with:

- sequential keys
- wide keys
- duplicate keys

Sequential keys are particularly bad with an ISAM structure, because the index is generated based on existing data values. All values greater than the existing largest key will be overflow pages. Hashing is good for this because values are distributed randomly throughout the space.

Wide keys are generally a poor idea. For ISAM and BTREE, this results in very large indices. The depth of an index grows logarithmically with the width of the key. For a hash structure, this does not increase the storage space, but does require more CPU cycles to evaluate the hash function. There is a trade-off on key size. A small key makes the depth of the index small, but is harder to make unique. Nonunique keys can increase the amount of overflow pages.

A high number of duplicate ISAM values for keys is also a problem in most storage structures. For an ISAM structure, this can result in a large number of overflow pages, since an index can only point to a single page. For a hashed structure, there are also many overflow pages caused by duplicate values.

Another form of excessive duplicity is a high number of nulls in a key column. A null value for a datum indicates that the data have no value. An example of this is a purchase order application where purchase numbers are not assigned for unapproved purchase orders. Each of these unassigned numbers has a null value, leading to many duplicates in the key value.

Horizontal decomposition is one technique to solve this problem. This involves putting two tables together, one for assigned purchase orders and one for unassigned. A view can be constructed that puts the two tables together as a single virtual table for the users. The unassigned table can be a heap structure, and the assigned table can use an indexed or hash structure.

The other solution is to add an extra column to the key if it is an ISAM structure. The key could thus consist of purchase order plus client name. Access by purchase order number is still efficient since the key search begins on the left side of the key. Finally, it is possible to use a random number instead of a null to reduce duplicates. For example, a null social security number can instead be some random negative number.

The question of the physical design of the database is often a difficult one for a multiapplication database. If the type of usage is always consistent—names always accessed by social security number—physical design is quite straightforward. Most environments, however, consist of a variety of different types of access patterns. The advantage of physical and logical separation of design is that performance can be modified and tuned based on evolving access patterns.

Query Execution Plans

The INGRES query optimizer is responsible for choosing the best method of getting data from database tables and combining the data into the results requested by the query. The resulting query execution plan is sent to the query executor, which issues calls to the Data Manipulation Facility to access the files containing the data.

The INGRES query optimizer takes an incoming query and chooses the best access plan—what order to access tables in and whether to use primary or secondary indices. To decide what possible access plans are for the data, the optimizer looks at the storage structure of tables, the presence of secondary indices, and the availability of statistical information on the data in the table (see Fig. 5-5). The optimizer then chooses the plan that appears to provide the most efficient access to the data requested and sends it to the query executor.

Query execution consists of taking the base data and combining them into a series of temporary tables. Note that the temporary tables could simply be a sort area in main memory, or if the data are very large, could be a table stored in a file on disk. Eventually, those temporary tables are all combined to form the results table. The query optimizer is responsible for deciding exactly how to combine these tables together.

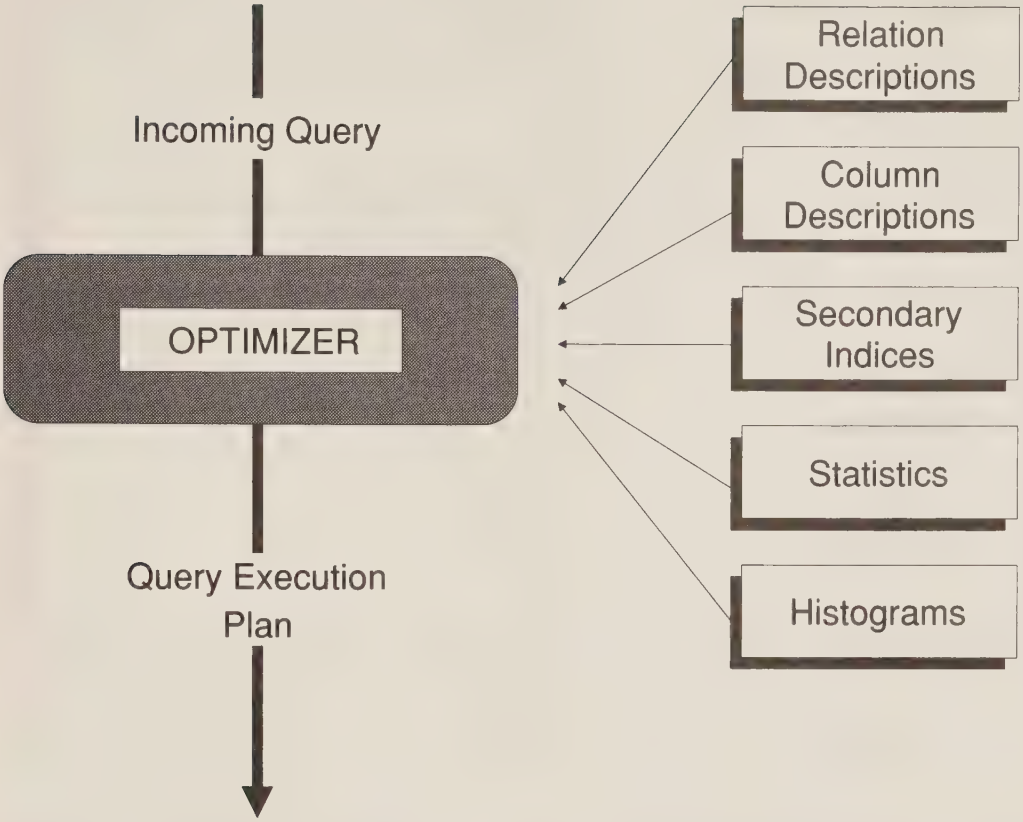


Fig. 5-5 Information Used for Query Optimization

One of the unique features of INGRES is that users can observe the process that the query optimizer will go through. Normally, users will simply want their data. When a query runs slowly, however, it is worthwhile to run the query through the terminal monitor and observe the query execution plan (QEP).

A QEP consists of a series of base tables, called leaf nodes, at the bottom of a hierarchy. Each node of the hierarchy at higher levels represents a particular type of operation, which then results in a temporary table. Figures 5-6 and 5-7 show a QBF query and the QEP associated with that query.

A leaf node is represented in a QEP by three types of information:

- the name of the table
- the storage structure
- the number of pages and rows (tuples) in that table

The storage structure for a table also includes the key for all nonheap structures. For some queries, it is possible that the key is listed as "NU," meaning that the index is not used and a scan of the base table is to be done.

All nonleaf nodes on the QEP include either three or four rows of information:

- the type of operation used to create this temporary table
- the storage structure of the temporary table (not present on all operations)
- an estimate of the number of rows in the temporary table
- an estimate of the cost of creating the temporary table

By default, all temporary tables in INGRES are created as a heap. It is possible for special situations to change this default operation.

The number of rows and the cost for creating this temporary table are estimates because the data are not actually retrieved yet. Remember that a relational system expresses queries in a logical fashion. The query

```
select * from emp where age=999
```

could theoretically retrieve every row in the database; the optimizer knows nothing about employee ages. The optimizer has to make a guess as to how many rows the equality will match. As a general rule, the equality is considered to be a restrictive query and will match only a small percentage of the rows in the table.

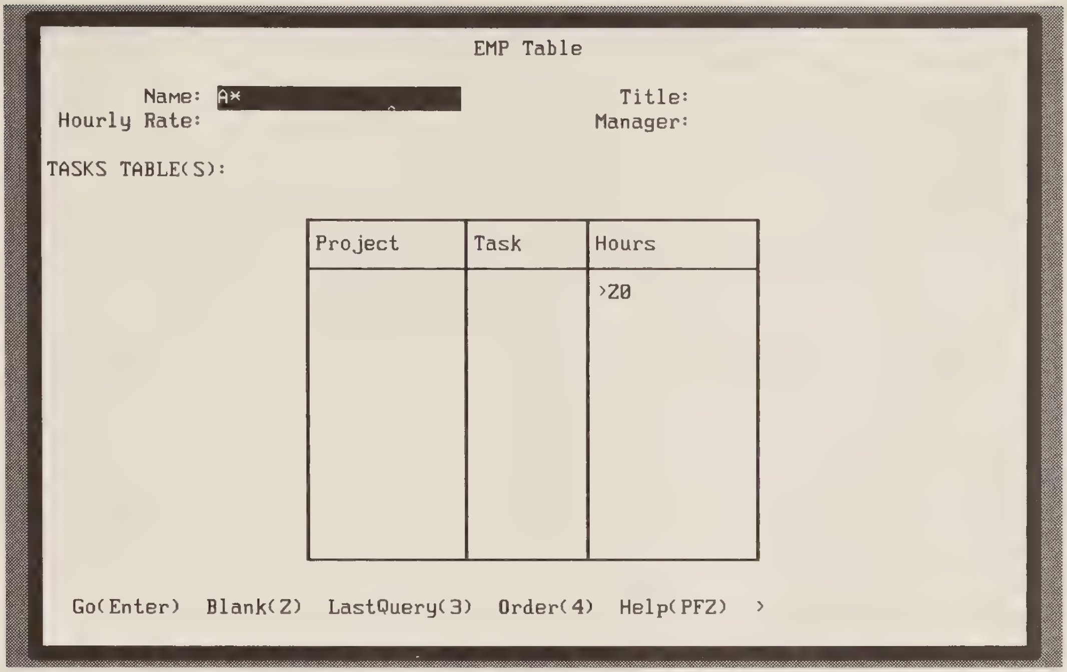
For purposes of a QEP, we can only estimate the cost and the number of rows. The number of rows is expressed using two numbers, the number of pages and the number of tuples (rows). The cost of the query is expressed as a function of disk I/O and CPU resources. I/O is expressed as the number of 2048 byte pages that are retrieved, which is the unit of I/O that INGRES uses. CPU is a metric that can only be used in comparing different query execution operations to each other.

Every node in the QEP represents the cumulative cost. To calculate the cost of each operation it is necessary to subtract the figures for the nodes directly below the operation.

The type of operation used to create a temporary table is really the crucial information. A cartesian product, for example, is a combination of every row in one table with every row in another table. With two 10,000-row tables, the result is a 100,000,000-row table! If we're trying to perform a join with the following qualification, the cartesian product is not desirable:

```
select * from emp where emp.name=mgr.name and mgr.name="JONES"
```

The usual reason for reading a QEP is to look for just this situation. The solution is fairly obvious in this case—create a primary or secondary index on name for both tables.



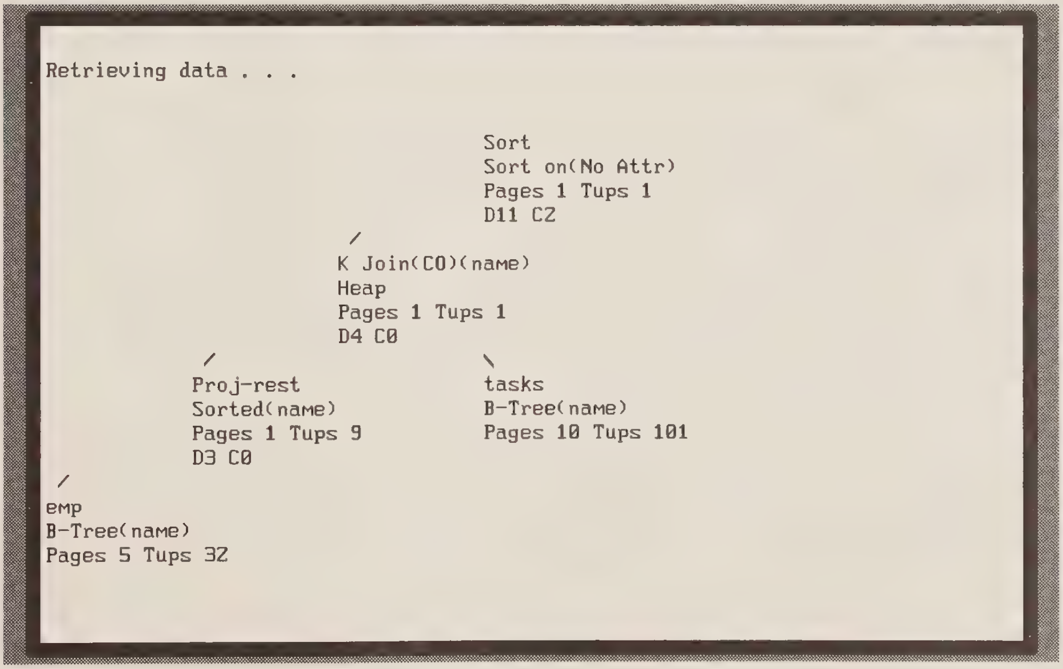
```
                                 EMP Table

        Name: A*                                   Title:
 Hourly Rate:                                    Manager:

TASKS TABLE(S):
```

Project	Task	Hours
		>20

```
  Go(Enter)  Blank(2)  LastQuery(3)  Order(4)  Help(PF2)  >
```

Courtesy of Relational Technology

Fig. 5-6 A QBF Query Using a Joindef



```
Retrieving data . . .


                                    Sort
                                    Sort on(No Attr)
                                    Pages 1 Tups 1
                                    D11 C2
                        /
                        K Join(CO)(name)
                        Heap
                        Pages 1 Tups 1
                        D4 C0
                /                   \
            Proj-rest               tasks
            Sorted(name)            B-Tree(name)
            Pages 1 Tups 9          Pages 10 Tups 101
            D3 C0
 /
emp
B-Tree(name)
Pages 5 Tups 32
```

Courtesy of Relational Technology

Fig. 5-7 The Query Execution Plan for the QBF Query

In addition to the dreaded cartesian product, there are three other types of operations that produce temporary tables:

- sort nodes
- project-restrict nodes
- join nodes

An ordering clause is an obvious reason for a sort node. Whenever a query has an *order by* clause, the top node of the QEP will be a sort clause. As a general rule, sort nodes mostly consume CPU resources. This is because the data are loaded into a sort buffer. The buffer is then used for the sorting operation. Only when the data do not fit into the sort buffer does this type of operation result in a large number of I/O operations. The size of the sort buffer can be set as needed.

A project-restrict node is used to remove certain columns or rows that are irrelevant to a query so that subsequent operations (such as a sort) are not forced to carry around excess data. Any column not referenced in a where clause or in the target list of a query would be removed with this type of node. Those columns that are retained are projected. Any rows not satisfying the where clause would be restricted.

Join nodes are where two tables are combined together. The query optimizer uses several different types of join strategies depending on the characteristics of the underlying data and the query that was specified. A sort merge is a join strategy that takes two tables, sorts them, and then performs a join row by row. This is opposed to a hash join strategy, which performs a lookup on the rows of one table instead of the sequential access of the sort merge.

There are two kinds of sort merge strategies. A full sort merge requires both tables to be sorted before the join can occur. This would be the case when the primary storage structure of the table is hashed or heap, both unsorted storage structures. The full sort merge can be recognized on a QEP by the presence of three types of nodes: two sort nodes and one join node (see Fig. 5-8). A partial sort merge consists of only sorting a single side of the join. This occurs when one of the relations is already sorted, as in the case of a BTREE.

If a retrieval consists of joining two tables, the QEP might consist of the following steps (see Fig. 5-9). First, at the bottom of the tree would be the two leaf nodes. Next, there would probably be two project-restrict nodes, one for each of the base tables. Then, if both tables are unsorted, the QEP would consist of two sorts and a final join to produce the desired results.

Secondary indices are not always used by the query optimizer. If a large percentage of the rows in the base table is to be accessed, it is easier to skip the additional overhead of going to the secondary index. One of the primary purposes of examining a QEP is to see if a secondary index is in fact used. When a secondary index is used, the join field is on the TID. The secondary index consists of a series of key values and a TID pointer. This pointer is a one-to-one mapping to a particular row in the base table, identified by the TID.

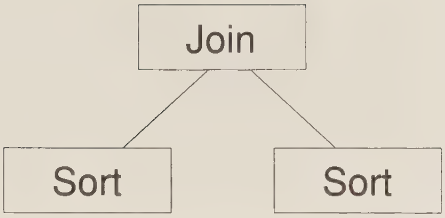


Fig. 5-8 Full Sort Merge Join Strategy

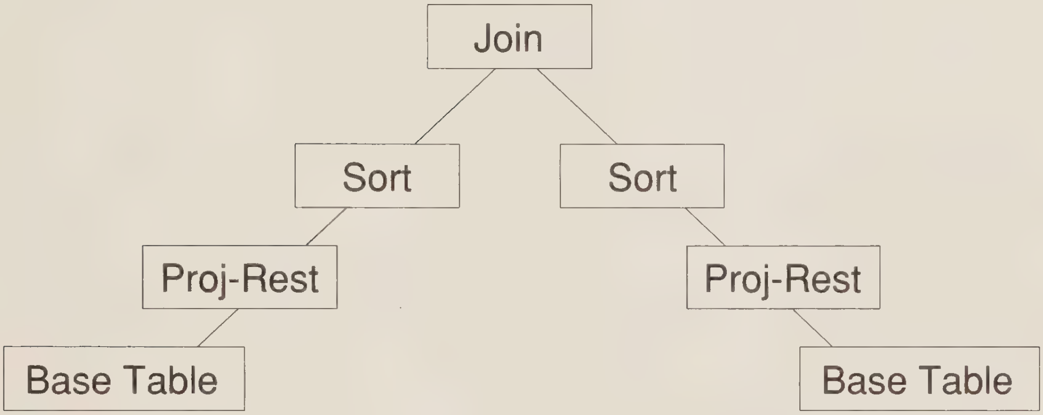


Fig. 5-9 Possible QEP for Joining Two Tables

A secondary index is treated just like a base table in the QEP. The index is a relation, which is first projected and restricted. It is restricted based on the where clause in the query. It is projected by only returning the TID pointer. The resulting temporary table is then sorted, and then joined to the base table.

A similar lookup uses the primary storage structure of a table to look up a series of values. Those are also restricted and sorted, then joined to another table. Primary index lookups are performed when the QEP estimates that relatively few rows will be returned, making the keyed lookup on the two tables more efficient than a sort merge strategy.

A cartesian product consists of a combination of every row in one table to every row in another table. This is not efficient and should be avoided wherever possible. A frequent cause of a cartesian product is a query that does not have a where clause to join the two tables:

```
select e.name, t.task from emp e, tasks t
```

This query was probably meant to retrieve all tasks that a particular employee worked on. As written, however, it retrieves every possible combination of name and task. The proper syntax for this statement should instead be:

```
select e.name, t.task from emp e, tasks t
        where e.name = t.name
```

The query optimizer does not usually list what type of join strategy is used, except for the cartesian product. For other types of joins, it is necessary to examine the underlying nodes in the hierarchy to determine which strategy was chosen.

The query optimization process consists of examining a wide variety of possible execution plans. It is possible to examine so many plans that it would have been quicker to just go to the base table and get the data. If the query optimizer sees that too much time has been spent looking, it times out and takes the best plan it has arrived at so far.

For most situations, the time-out situation is not reached because most queries are fairly straightforward. It is only on highly complex queries with a large number of tables and aggregates that time-out becomes a factor. It is possible to tell the query optimizer not to time out and keep looking for the optimal plan.

When a query is repeated many different times, it does not make sense to optimize queries each time. A precompiled query consists of a saved QEP. The query processor is able to take this saved plan and directly execute it. A *repeat* operation in INGRES is an example of a stored QEP. When an SQL operation is sent to the back end, the user can signify that this operation will be executed several times by putting the word "repeated" along with the basic operation (select, update, insert, or delete). The data manager then saves the QEP for reuse.

Another precompiled operation is the database procedure. A procedure is a collection of SQL and control statements stored in the database. When the user says "execute" the procedure, the collection of statements is processed, and the result returned to the user interface. Procedures are precompiled and the query execution plan is stored in a system catalog.

Procedures thus have two types of performance gains. First, the QEP is already prepared. Second, the query itself is already stored in the system catalogs, meaning that the SQL statements do not have to be transmitted to the data manager. Instead, just the name of the procedure and parameters are sent through. In a networked environment, this can increase the throughput and performance dramatically.

Optimizedb

The query optimizer makes some assumptions about how much data will be retrieved based on the join operators involved. For example, in the following qualification, we assume that one row of a small table will join to a small percentage of the rows in a big table:

```
select p.product from product p, orders o,
      where
              p.product = o.product and
              p.category = "NEW" and
              o.amount > 20000
```

Optimizedb allows the query optimizer to make more informed decisions on the distribution of the data. Instead of assuming that a small percentage, say 1 percent, of the rows in the order table will match a product in the product description table, the query optimizer can examine two special system catalogs to look at a profile of data in the database.

Optimizedb is a utility run by the database administrator that examines the data in selected columns of a table and puts a profile of that data into the system catalogs. Optimizedb is usually only run on columns that will be involved in a where clause, which are typically primary and secondary keys for tables.

For very large tables, optimizedb can take a long amount of time to run. INGRES is able to sample data in large tables, thus greatly speeding up the performance of this utility.

Optimizedb should be run whenever the profile of the data in the database changes. If a large number of rows are updated, the profile of that column needs to be updated so that the query optimizer has an accurate picture of the data.

There are two levels of data that can be collected by optimizedb:

- basic statistics
- histograms

Basic statistics give a profile for the uniqueness and distribution of the data, including the number of unique values in a column and a repetition factor that indicates how many repetitions can be expected for different values. Instead of guessing that a join will match a small percentage of the rows in a table, the basic statistics can be consulted to get a more accurate estimate.

Knowing what percentage of the data might be retrieved helps the query optimizer in two ways. First, the optimizer can decide if the use of an index can be beneficial. Second, the optimizer can decide what order to join tables together. By looking at the number of rows in the table and the selectivity of the retrieval, the optimizer knows roughly how much data will be returned. By comparing the various estimates for the amount of data returned, the optimizer can select the best order for getting data.

The second statistics table is used to keep a histogram. Normally, optimizedb breaks the data into fifteen groups, or cells, in a histogram. It is possible to have histograms with up to 56 cells for highly distributed data. When a query is formulated asking for data in a certain range, optimizedb can get an estimate of the number of rows within that range.

The detailed data-profiling capability of optimizedb is a unique feature in the INGRES data manager. An accurate profile of data can greatly increase query execution time on complex queries. Instead of making an arbitrary estimate of the selectivity of a join or where clause, an accurate profile can be constructed that reflects the particular nature of that query.

Modifying the Query: Permits, Integrities, and Views

Three data manager facilities are used to modify a query received from the front-end process. Permits, integrities, and views are all transparent to the front-end process. The incoming query is modified to take into account these three factors, and then submitted to the query optimizer. It should be noted that al; three of these facilities could be built into the front-end user interface. For example, a programmer can put an integrity check into a VIFRED form validation. The programmer can also check the user name and restrict access to certain forms of data.

The problem with moving these facilities into the front end is that only certain front-end programs have them. If a programmer builds in password protection into a custom application and the user then uses QBF to access data, the password protection is bypassed. Building these facilities into the data manager means that a wide variety of user interfaces, including the general-purpose facilities of VIGRAPH, RBF, and QBF, can be used to access data, allowing for more flexible, yet still secure, access to information.

In INGRES, data that are shared among multiple users must be owned by the database administrator (DBA). Private tables, created by individual users, are only accessible to that user. Tables owned by the DBA are made public through the use of permits. Permissions in INGRES operate at two levels:

- environmental permits
- data-valued permits

The query is received by a process in the data server called QRYMOD. QRYMOD first checks the environmental permits, and if those constraints are satisfied, it modifies the query to add on the second level of constraints. Note that in SQL, only environmental permits are allowed. To restrict access to data in SQL based on the value of a certain column, the database administrator would first create a view that contained the restriction in the where clause. Certain classes of users would then be given permission to access the view. Since the QUEL syntax is more robust, it is used for the security examples in the following discussion.

An environmental permit is used to restrict access to certain tables or columns of a database by user name, the terminal type, or the time of day. The operations available are appends, replace, retrieve, delete, or all. An example of an environmental permit on data is

```
create permit update on emp (salary)
        to amartin
        at tta0:
        from 9:00 to 17:00
        on Monday to Friday
```

This permit requires that the user *amartin* be on the terminal represented by the device *TTA0:*. This is useful to prevent somebody who accesses the database from another device, say a dial-in user accessing the system via modem, from gaining access to sensitive tables.

The time and day restrictions are also useful for restricting operations on sensitive data. As a general rule, one can presume that updates to the salary database will not occur at 11 at night or on weekends. This environmental restriction prohibits the updates of the salary information except during normal business hours.

A data-valued permit restricts access to information based on the value of the data. This is equivalent to adding a where clause to every query that falls within the scope of the security constraint. For example, the following permit could be defined:

```
create permit update on emp (salary)
        to amartin
        where emp.salary < 100000.00
        and emp.position not in ( "Director", "President" )
```

This permit allows user amartin to update the salaries of all people who do not meet the criteria of making more then $100,000 and hold the position of director or president in the corporation.

Violations of environmental permits cause an error message to be returned to the user. Violations of data-valued permits, however, are not necessarily apparent to the user; the user will not know if rows were eliminated based on the original where clause in the query or the modified where clause created by the permit. It is possible that some of the rows found were eliminated by the modified version of the where clause. The user just sees fewer rows returned (or none if all are eliminated).

There are three system catalogs involved in the security framework. The *II_TABLES* catalog contains two bits used to indicate the presence of the "all to all" or "select to all" permissions. These blanket permissions are stored along with the definition of the table to allow efficient access to data. For permissions that are more complex, the query processor goes to a special permit table that contains permitted operations and environmental constraints. The data-valued permits are contained in a parsed query tree in a third system catalog.

It is possible for an infinite number of permits to be accumulated on particular tables. It is even possible to have duplicate permits on a particular table. It is up to the DBA to periodically purge unnecessary access constraints. Because there are multiple permits, the QRYMOD process needs to evaluate all of them to see which ones apply. The basic rule of thumb is that the broadest permit will apply. If there is a highly restrictive permit (no access to a table for a particular user) and a broad permit (permit all to all), the user will be allowed to access the data.

To evaluate a query, the QRYMOD process breaks all columns involved in a query into four classes:

- add/change
- retrieve
- aggregate
- qualification

Columns that form part of the qualification are evaluated separately from those in the retrieve columns. In both cases, the user needs retrieve permission, since a where clause is really a retrieval of data values. The implication of this separation is that is possible to allow a user to update a column without seeing the value. For example, a user could be given retrieve permission on name and update permission on salary.

Within each class QRYMOD looks for any one single permit that would allow the retrieval. In other words, QRYMOD performs an "or" operation on the different queries. A query to select name and salary from a table would result in both columns being placed in the retrieve column. QRYMOD then looks for a single permit that allows both columns to be simultaneously retrieved.

Among different classes, QRYMOD simply looks to see if there is access allowed in both cases. For example, if a query retrieved name based on a particular value of salary, there can be two separate permissions. One permit allows retrieval on name, the other allows retrieval on salary. This permit structure has an interesting side effect. If a user needs to be able to update a table, she needs both update and retrieve permissions, because an update statement will typically have a qualification in the where clause such as *where name = "amartin"* that is evaluated separately.

Aggregates are processed as a separate query with the results, then placed as part of the retrieve set of the main query. Each of the components of the aggregate must meet the permission structure in the database, as well as the outer query containing the aggregate.

INGRES has a very granular access structure based on individual user names. However, INGRES has no concepts of groups of people. This is somewhat inconvenient in situations with large numbers of users. A simple way to work around this is the following. A groups table can be created with two columns, user name and group. Then, permits are defined as:

```
create permit all on emp
        where username = group.name and group.group = "admin"
```

This permit allows anybody in the group "admin" to perform any operations on the emp table. The clause *username = group.admin* checks to make sure that the current user is in the admin group. If the group table is frequently accessed, it will stay in the cache and will thus not decrease markedly the performance of the query.

Because permits are added onto the end of a query, it is important to take into account the performance implications of data-valued queries. These can decrease the performance of the query, cause cartesian products, and even fill up the query buffer. Permits thus perform an important function, but at a cost.

The second type of query modification is the integrity. The permits operate on the basis of users or other environmental factors. An integrity operates on the basis of data values and applies to all users. An example of an integrity would be:

```
create integrity on emp is
                   date_hired > "June 1, 1980"
        and
                   date_hired <= "Today"
```

This integrity requires that the date hired field fall somewhere between the date of the founding of the company and today's date.

Integrities can also be used to match values from a list. For example, a purchase order can have a type of either "EXTERNAL" or "INTERNAL." The integrity for this constraint would be:

```
create integrity on purchase is
                   type in ("EXTERNAL", "INTERNAL")
```

Just because an integrity is defined in the database doesn't mean that VIFRED validations should necessarily be bypassed. It makes more sense to correct an error immediately than to have the user attempt to save the data and receive an error message back. A VIFRED validation check has two important advantages over a data manager integrity definition. First, the help screen may display a list of valid values when the user hits the help key on that field. Second, the forms designer can specify the error message that is seen by the user. When a VIFRED form is designed, it would also make sense in this example to force uppercase on the field type. If a user enters "External," it would automatically be converted to "EXTERNAL."

Front-end and back-end integrity checks thus serve complementary purposes. The VIFRED validation (or the INGRES 4GL code that performs a validation) can be used to correct errors as they appear on the user's screen. The back-end integrity constraints can be used to make sure that users of other applications, such as QBF without the VIFRED form, also observe the integrity of the data.

The third type of query modification is the view. A view, as discussed earlier, is a virtual table. When a user performs a select statement on a view, that query is added to the view definition. This modified SQL statement is then processed for integrities and permits. Finally, the modified query is submitted to the query optimizer for execution.

Views are often used to simplify the database structure for specific types of users. For a variety of design reasons, the database may consist of a large number of tables, some with columns not needed by certain classes of users. A view can combine tables and eliminate columns. Another use of a view is to make a general-purpose table serve specific classes of users. For example, a corporation could have a master sales table. Sales managers for particular regions would only be interested in the rows of the table that fall within their regions. Salespersons would only be interested in the rows in their sales areas.

The following two views help solve this problem:

```
create view region ( person, item, sale_value)
        as select  b.person, b.item, b.sale_value
                   from base_sales b
                   where b.manager = user

create view individual ( item, sale_value )
        as select b.item, b.sale_value
                   from base_sales b
                   where b.person = user
```

Both view definitions substitute the user's name into the query, and execute it. A salesperson could then execute the following select statement:

```
select * from individual
```

More likely than directly submitting an SQL statement, the salesperson would use a VIFRED form on the view "individual." The combination of form and query target would be a QBFname. The salesperson would simply select that query target from the QBF catalogs.

Views are also often used to create summary information. Total sales by region could be retrieved using this view definition:

```
create view total_sales ( region, sales )
        as select region, sum(sales)
        from base_sales
        group by region
```

Every time a select is run on the total_sales table, the sum of sales by region is recalculated. The advantage of this approach is that the summary sales figures are always up to date because they are derived each time. The disadvantage of creating views for aggregate data is that this particular query could involve quite a bit of processing. If the view is accessed frequently, there is a lot of duplicate processing done.

Another strategy is to have another table, not a view, called total_sales, which contains the same information. This approach requires that the derived table be periodically updated. Usually, this would be done hourly or nightly. Right after the table is updated, it has the most current information. As time passes, if there are updates to the base_sales table, the summary table will become out of date.

Instead of recalculating the table periodically, another approach is to have the front-end application recalculate the information every time it changes the base table. This means that QBF would not be an appropriate tool for updating information (it would still work for retrievals). A custom application would have to be written using ABF that performs both updates when a user enters a new sales item. This approach would obviously slow down update operations.

The use of views versus derived tables has important implications for the performance of the system and the currency of the data retrieved. Again, it is important to remember that users do not see these issues. When they request data from the total_sales query target, it does not matter if that object is a view or a derived table.

Summary

This chapter examined issues on the physical design of a database, including the storage structure of a table, the choice of a primary key, and the use of secondary indices. In addition to modifying the tables containing data and creating secondary indices, the DBA is able to add permits, integrities, and views that change the operation of the database.

When a query is received by the data manager, it is first modified, then optimized. The result of the optimization is a QEP. This plan then requests low-level data using various access methods from the DMF. DMF, in turn, uses the services of the operating system to perform low-level access to the files that contain the data.

In this chapter we have ignored the nature of the system that INGRES runs on. We have also ignored the situation of multiple users accessing data from the same table. The operations in this chapter are the same in all of the INGRES environments, be it a MicroVAX or a large IBM mainframe.

In the next chapter we begin to consider the multiuser environment. Locking of data, archiving data, and recovery from system crashes are some aspects of this issue. Another aspect is management of data servers and disk drives on a computer to optimize performance.

Chapter

6

Multiuser Data Access

INGRES and Computer Configurations

Until this point, we have looked at an environment consisting of a single user interface working with a single data server. In this chapter, we move those two processes into a real computing environment consisting of multiple users sharing common access to a database.

One of the jobs of INGRES is providing transparent access to the operating system and file system of a computer. Users see SQL statements, and possible Query Execution Plans (QEPs). Ultimately, the Data Manipulation Facility translates these requests for data into a series of low-level calls that are sent to the file system, which in turn extracts the data from disk.

Conceptually, a computer has four components. The CPU is what does the actual processing. CPUs are often measured in terms of millions of instructions per second (MIPS). It is important to take this measure of processing power with a grain of salt: different computers have different kinds of instructions. For example, a VAX minicomputer has a relatively complicated instruction set. With relatively few instructions the VAX can accomplish quite a bit of work. A Sun 4 computer, by contrast, is a reduced instruction set computer (RISC). Each instruction is extremely quick, but it could take more instructions to accomplish the same amount of work.

MIPS are useful for comparing computers of the same architecture, such as different VAX systems. Precise comparisons among different machines are not effective. MIPS are used often as a broad measure of power—a 200 MIP Cray is definitely more powerful than a 10 MIP VAX.

Another reason that MIPS are not necessarily useful is that users don't really care how many instructions are performed by the CPU; they care about the throughput of their application. In order for the CPU to process information quickly, it has to be able to get the data and return the results. Performance on a system is thus a function of the balance of many key components. From a hardware standpoint, the CPU is supplemented by three other important components:

- memory
- a bus
- disk drives and controllers

Before the CPU can process data, they have to first come off of the disk drive. If the disk drive is saturated, it does not matter how quickly the CPU operates! Once the data are read off the disk, they have to travel over a bus into main memory (see Fig. 6-1). Main memory then serves as a staging area for processes that are ready to compute. A balance among all these components is essential.

The operating system manages these different hardware resources and makes them available to users in a multiuser computing environment. Running on the operating systems are a variety of different programs. The INGRES data manager and user interfaces are two such programs.

This chapter will examine how INGRES uses the services of the operating system to access data and guarantee their integrity. If two users are changing the same datum simultaneously, inconsistent results are highly probable. One portion of the INGRES data manager, known as the locking system, coordinates the access to data on a disk drive.

INGRES can be configured for different machines to effectively use main memory and disk drives. When a system manager installs INGRES, she specifies how INGRES will use these different services to optimize performance. For example, we will see that distributing data among multiple disk drives is one way that INGRES can more effectively use the services of the operating system and the computer.

Servers

In INGRES, access to a database is via a server. This database server is able to access several different databases, and a server can process requests for several different user interface processes. In many older relational database systems, each user has his own server process. Coordination among these different back-end processes is done through the locking mechanism. The problem with this architecture for database access is that each user has two processes: a front end and a back end.

In a lightly loaded system, a front- and back-end process for each user does not matter. However, when a system gets heavily loaded, the processes start competing for memory. Since not all processes can fit in memory at the same time, some of the front- and back-end processes are swapped out: the entire process is taken out of memory and put on a disk drive. When the system load lightens up, the swapped-out processes are swapped back in. Swapping is very inefficient because the system must go to the disk drive to find the process and load it back up before it can continue operation.

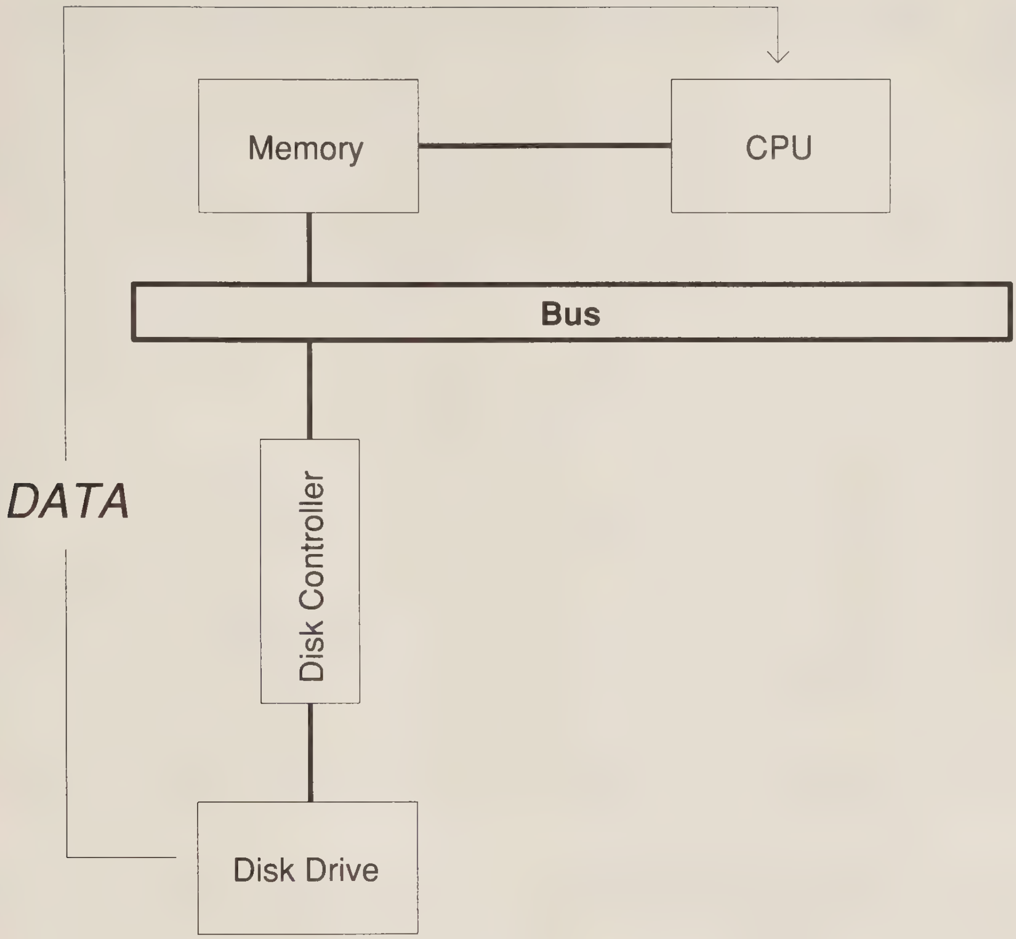


Fig. 6-1 Components of a Computer System

INGRES has a server architecture in which a single back-end process is able to service many different users. Instead of each user having his own back-end process in memory, the services of a data manager are shared. By sharing this resource, more efficient use of memory is made and INGRES is thus able to service a larger number of users.

The INGRES server architecture has two other implications. First, INGRES allows multiple servers to access the same database. In a loosely-coupled environment, such as a VAX Cluster, this allows multiple CPUs to be used for the database application, increasing overall performance for data access. It is also possible to have several servers on a single-processor computer. Multiple servers on a single computer might be configured so that one server runs at a higher priority for a specific group of users. The second implication of a server architecture is that the server can coordinate different users requests more effectively. As will be seen, the server can group transactions together and commit them all at the same time. Since a disk drive is limited in the number of I/O operations per second it can perform, grouping transactions allows a single I/O operation to service multiple users.

When a server is initialized, the system manager can configure that server in a variety of different ways. Most servers are public and can accept requests from any authorized INGRES user. Sometimes, a server can be made private for particular users. Another option is to make a server only accept requests from a group of people.

A server, like any other process on a computer, runs with a certain level of system resources that it is authorized to access. For example, the server might run at a certain priority, giving it access to the CPU before users with a lower priority. The server is also given a quota on the amount of main memory it can use.

One advantage of multiple servers, particularly private servers, is that they can be installed with higher access to resources than the public server. Users that are allowed to access these private servers will get higher performance than others. For example, a data entry application might be configured to use a private server since this is a fairly crucial operation. Generic ad hoc queries, on the other hand, would go to the public server.

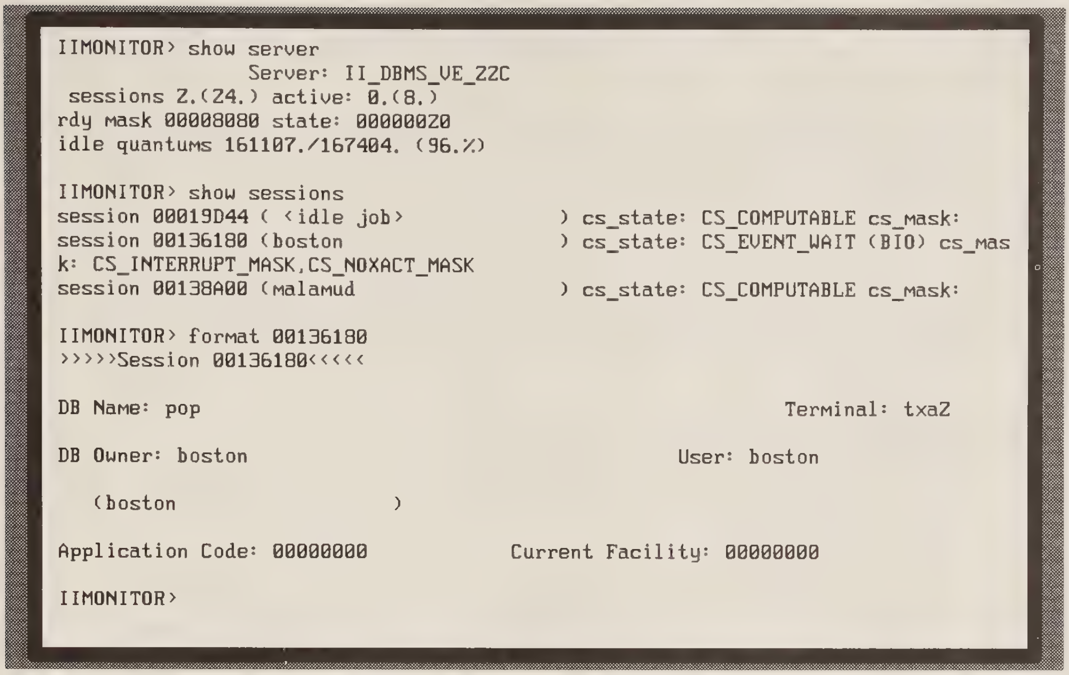
In addition to the access characteristics of a server, it is possible to configure the maximum number of connections that the server will accept. This is useful so that the server does not accept so many connections that users see slow response time. It is usually desirable to turn down the next connection request instead of degrading performance to an unacceptable level for all users.

To examine servers on a system or sessions on a particular server, the system manager can use the IIMONITOR utility (see Fig. 6-2). Overall parameters for the server show the total number of sessions compared to the total number allowed, as well as the number of active sessions.

The system manager can also examine the status of all sessions using the *show sessions* command. Sessions in a server can be in one of three states. A CS_EVENT_WAIT state means that the front end is waiting for some event to occur. The LOCK event state means that the user is waiting for a lock to become available. A CS_COMPUTABLE session state means that the process is awaiting execution. For example, a query may have arrived and the server is required to parse and optimize that query.

To examine a particular server session, the system manager issues the *format* command along with the identification number of the session. The display shows which terminal the application is running on, what database it is using, and the owner of the database. It also shows which INGRES facility is being used for the current access.

The IIMONITOR utility also allows the system manager to stop a server. The *stop server* command immediately stops the server, aborting all ongoing transactions. The *set server shut* command disallows new connections, shutting the server down when the current sessions exit.



```
IIMONITOR> show server
                Server: II_DBMS_VE_22C
 sessions 2.(24.) active: 0.(8.)
rdy mask 00008080 state: 00000020
idle quantums 161107./167404. (96.%)

IIMONITOR> show sessions
session 00019D44 ( <idle job>             ) cs_state: CS_COMPUTABLE cs_mask:
session 00136180 (boston                  ) cs_state: CS_EVENT_WAIT (BIO) cs_mas
k: CS_INTERRUPT_MASK,CS_NOXACT_MASK
session 00138A00 (malamud                 ) cs_state: CS_COMPUTABLE cs_mask:

IIMONITOR> format 00136180
>>>>>Session 00136180<<<<<

DB Name: pop                                                 Terminal: txa2

DB Owner: boston                                    User: boston

   (boston                  )

Application Code: 00000000            Current Facility: 00000000

IIMONITOR>
```

Courtesy of Relational Technology

Fig. 6-2 The IIMONITOR Utility on a VAX

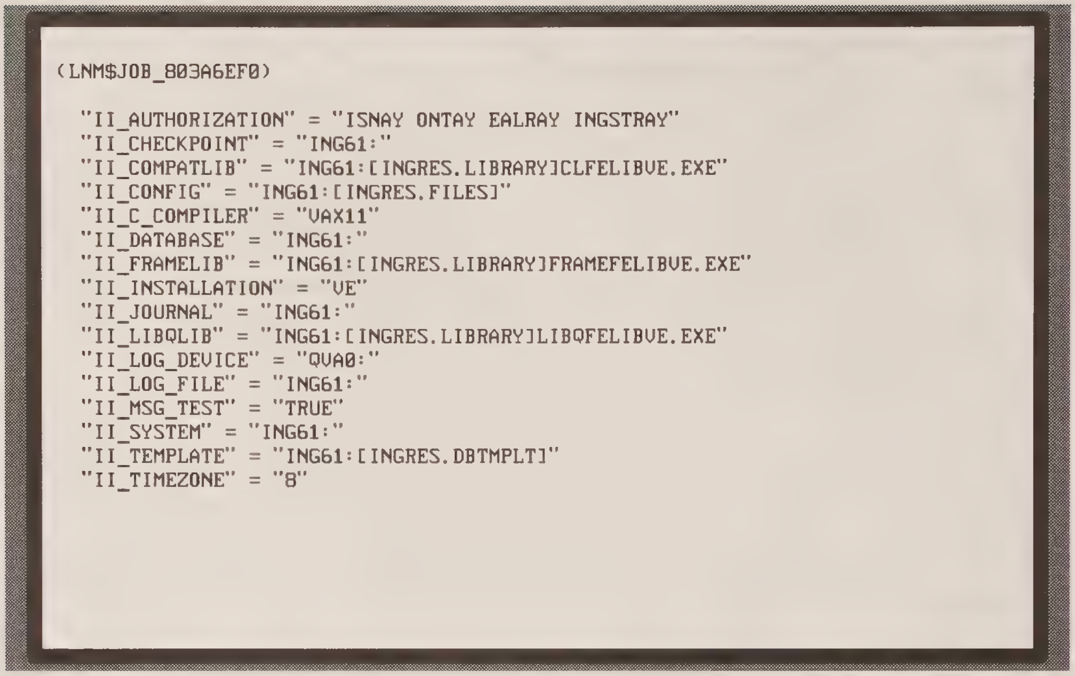
Environmental Variables and Locations

An INGRES environment consists of a large number of files, including files containing the actual data, files that hold the programs for utilities such as QBF, user files, and a variety of files that contain consistency information, such as logs and journals. These logs and journals, discussed later in this chapter, guarantee that even if the database is damaged, it will be possible to restore the database.

One job for the INGRES installer is to determine which files go on which disk drives. It would be easy to put all files on one disk drive, but this has several potential problems. If a log, which is there in case data get corrupted, is on the same drive as the data it does not do much good when the disk drive breaks.

Another problem with putting all data on one disk drive is that they just may not fit. A database of several gigabytes of data will have to go on several different disk drives. Even if the database would fit on a single disk drive, this can create a potential bottleneck. No matter who is accessing what data, all users eventually are put into the queue for a single disk drive. Since a disk drive is limited on its potential throughput, this drive then becomes a bottleneck on the system. There may be plenty of memory and CPU capacity, but the system is limited to the performance of the drive.

On most operating systems, such as VMS and Unix, INGRES uses environmental variables to decide where certain types of data go. An environmental variable, such as a



```
(LNM$JOB_803A6EF0)

  "II_AUTHORIZATION" = "ISNAY ONTAY EALRAY INGSTRAY"
  "II_CHECKPOINT" = "ING61:"
  "II_COMPATLIB" = "ING61:[INGRES.LIBRARY]CLFELIBVE.EXE"
  "II_CONFIG" = "ING61:[INGRES.FILES]"
  "II_C_COMPILER" = "VAX11"
  "II_DATABASE" = "ING61:"
  "II_FRAMELIB" = "ING61:[INGRES.LIBRARY]FRAMEFELIBVE.EXE"
  "II_INSTALLATION" = "VE"
  "II_JOURNAL" = "ING61:"
  "II_LIBQLIB" = "ING61:[INGRES.LIBRARY]LIBQFELIBVE.EXE"
  "II_LOG_DEVICE" = "QVA0:"
  "II_LOG_FILE" = "ING61:"
  "II_MSG_TEST" = "TRUE"
  "II_SYSTEM" = "ING61:"
  "II_TEMPLATE" = "ING61:[INGRES.DBTMPLT]"
  "II_TIMEZONE" = "8"
```

Courtesy of Relational Technology

Fig. 6-3 INGRES Logical Names on a VAX

VMS logical name, points to a device on the system. For example, when looking for the default location for databases, INGRES first consults the logical name *II_DATABASE*. II_DATABASE then points to a device, say *DUA0:*.

On some operating systems, it is possible to have several levels of logical name translation. II_DATABASE might get translated into a device ING61:, which in turn gets translated into a physical device DUA0:. Translating twice means the system manager can move the data to a new device (i.e., DUA1:) without reinstalling INGRES. This makes the installation more portable. Figure 6-3 shows a portion of the logical name table on a VAX with the VMS operating system that uses two levels of translation for the II_DATABASE logical name (note that the translation of ING61: to a physical disk drive is not shown in the illustration).

An example of the advantage of two levels of translation is the case of a disk drive hardware failure. Presumably, the system manager has a backup of the database. The manager can put the data on DUA1:, and change the logical name DISK1: to point to DUA1: instead of DUA0:. INGRES continues to operate unchanged because it always looks for data on the logical name DISK1:.

Logical names are typically put into a log-in file that is executed for every user who logs onto the system (or every user in a certain group). This is known as a system logical name. It is entirely possible, indeed common, to have users override the II_DATABASE logical name and provide their own translation. An example would be a special-purpose database that does not fit onto the default area for INGRES databases.

Another example is a programmer that wants to test changes in applications, but does not wish to do so in the production environment.

Several logical names are used for different types of INGRES files. *II_CHECKPOINT*, for example, is used to find the location of the disk drive that stores backup copies of the database. Needless to say, it makes sense to store this information on a different disk drive than II_DATABASE! Another logical name is *II_SYSTEM*. This is the device where the programs that INGRES uses are stored. On a large installation, it makes sense to put these files on a different disk drive than II_DATABASE. That way users accessing programs (i.e., QBF) will not be competing for disk drive capacity with users that are accessing data. Multiple disk drives increase the bandwidth available for INGRES to use.

On most of the INGRES environment, there is a series of special files used by the operating system. For example, a paging file is used to hold parts of programs that don't fit into memory. It is important to put these system files on a different disk drive from the INGRES database in an environment with heavy data access.

Logical names are also used for INGRES locations. II_DATABASE is the default location for databases. As discussed earlier, it is possible to have II_DATABASE point to a different disk drive as the default location.

What about the situation where a single database (or single table) does not fit on a single disk drive? INGRES allows a single database to span multiple locations. The user first defines alternate locations using a logical name that points to another disk drive. Registering this alternate location (i.e., *EXCESS_DATA*) is then followed by extending a particular database, so that the data manager knows that the database is authorized to use the new space, by using the accessdb utility.

To create a table on a new location, the user simply adds a clause to the create table command that says *with location = EXCESS_DATA*. The default tables, such as system catalogs and tables created without a with clause, are put in II_DATABASE. The new table resides on the new location. INGRES adds this information to the definition of that table so that the data manager knows which disk to go to in order to find the file.

A multilocation table is a single table that is fragmented into several pieces, each piece in a different location. This is done for two reasons. First, the data may just be too big for a single location. The second reason is performance. Because a single disk drive can perform a limited number of I/O operations per second, this is an upper limit on the amount of data that can be accessed from a single disk drive. Even with caching, there is usually a limit that will be less than the processing capability of many CPUs. A multivolume table lets the table be split among multiple disk drives, allowing an increase in I/O throughput.

Environmental variables (logical names) are also used by INGRES for a variety of purposes in the front ends. For example, *TERM_INGRES* defines what kind of terminal the user has. When a user interface draws information on the screen, it is able to use TERM_INGRES (and a few terminal capability files and maps) to issue the proper command to drive that particular type of terminal.

Other front-end variables define how the user sees data. For example, *II_DATE_FORMAT* can be set to display dates in the U.S., Swedish, German, or other formats. *II_DECIMAL* can be set to make the decimal point a comma or a period. Using logical names means that the user interface can be different for different types of users without changing the front-end application for each situation.

An INGRES installation at this point consists of a server and a series of environmental variables that point to disk drives used to store the data. There are three other processes in an INGRES installation that will be discussed next. The lock manager ensures that incompatible operations on data by two users will not be performed. The recovery manager and archiver ensure that a system crash will not lead to an unusable database.

Locking

When the Data Manipulation Facility (DMF) requests data, it sends a series of read and write commands to the file system on a particular computer. This file system is also used by other applications, say a word processor or a mail utility. An important characteristic of a file system is that it is a general-purpose utility. Its sole purpose is to provide access to files. It takes a queue of requests and processes them one by one.

An INGRES table is an example of a file. Within that file may be index, leaf, and data pages. We saw in the discussion of the BTREE structure that an update to a data row can often result in updates to several index pages. Each of these operations is a separate write operation to a different page of data. What if another user is also updating a different row of data? That write operation could also update the same index pages. Without some coordination, these various write operations could all be interspersed in the same queue, yielding wildly inconsistent results.

The BTREE update situation is an obvious example of the need to protect or lock certain pages of data until an entire operation is concluded. Then, the next operation can proceed. Another situation where locking is needed is when the semantics of the operation require it. An example is moving data from a checking account table into a savings account table—a transfer of money from one account into another. The data manager has no reason to suspect that these two operations require locking.

Users can group these two operations into a multistatement transaction. This transaction tells the data manager that it must process both operations as a single transaction. Other users should not be allowed to access the checking account data until both parts of the transaction have been performed. Multistatement transactions thus ensure the integrity of several operations across time. Locking ensures that the consistency of operations at a particular point in time is preserved by providing an orderly access to data among multiple users.

This section examines how the data manager decides to lock data. Here we will focus on the question of locking granularity. Granularity is the issue of how much of a table to lock: whether to lock only a single page or to lock all the pages in a table. Locking the entire table allows a user to quickly access multiple rows without accessing additional locks, but it keeps other users from using that table for the duration of the transaction.

There are two general classes of locks. A shared lock allows multiple users to have the same lock. An example is when several users are only reading the data in a table. There is no need to exclude other users. The reason for the shared lock is so that no user tries to write data pages that another user is reading. The other kind of lock is an exclusive lock. When writing a data page, the data manager puts an exclusive lock on the data. The exclusive lock only allows a single user into that region of the database. If another user tries to get either a shared or exclusive lock on the same region, the request is either queued or denied.

In addition to the two classes of locks, locks can be requested at different levels. For access to data, locks are usually at the page or table level. In addition, it is possible to lock an entire database. If the system manager needs exclusive access to the database, he would request an exclusive database-level lock.

Some INGRES utilities automatically request a table-level lock. When a user is modifying the data structure of a table, she needs an exclusive lock on the entire table. Other utilities, as in the case of modifying the system catalogs, require an exclusive lock at the database level. For normal queries, it is the responsibility of the query optimizer to decide what level of lock to take on a table. For example, examine these two queries:

```
select * from emp
select * from emp where emp.ssn = "346526047"
```

Since there is no where clause on the first query, it will, by definition, involve a scan of the entire table. This would result in a shared read lock taken on the emp table at the table level.

The second query is probably highly restrictive. The query optimizer would examine several things to decide how many pages are likely to be accessed. First, the optimizer would look for the presence of a storage structure or secondary index that allows direct access to the data based on social security number. Next, the optimizer would look at the operator used in the where clause. By definition, an equal operator is fairly restrictive (as opposed to a greater than sign). Finally, the query optimizer would look in the statistics and histograms catalogs to get a more informed estimate of the probable effect of the query.

The rule of thumb for the query optimizer is that if 10 or fewer pages are likely to be affected by a query, a series of page-level locks will be taken. If more than 10 pages are affected, the query optimizer requests a table-level lock. It is possible to change the maximum locks parameter so that more than 10 page-level locks are taken before the lock is escalated to the table level.

The query optimizer can only make a guess as to the number of pages that will be referenced. In the course of performing the query, if more then 10 pages are actually used, the data manager escalates the lock level. It first requests a table-level lock, then relinquishes the previously-held page-level locks.

Locks are escalated because they are a limited resource on the system. Remember that all other queries must first obtain a lock from the lock manager before they can access data. If there are a large number of locks outstanding, there are many locks to check before granting access to data. It is not unusual for a transaction to be unable to get a lock immediately. In most situations, the wait will be minimal. There are three types of situations, however, where locks may not be granted:

- timeout
- deadlock
- livelock

These situations all occur where multiple users are accessing the same data.

Timeout is affectionately known as the "gone-to-lunch syndrome." When a user is in QBF, an update operation consists of a single transaction. Data are retrieved with a write lock and then examined by the user. Eventually, the data are saved back in the database. If the user goes to lunch in the middle of a QBF update operation, that entire table can remain locked (or series of tables in the case of a JoinDef or a view).

If another user is also trying to access the same data, she waits. The default in INGRES is that users wait forever. There is a timeout parameter that can be set for a session that instructs the data manager to only wait for a certain period of time for a lock, and then return an error message to the application. The application (or user) can then decide to try again.

Another example of a long wait period is when a large report is being run. Reports typically involve a shared table lock on a table or series of tables. All update operations on those tables would have to wait until the report is done running. This is fine for short reports. However, if the report involves sorting several gigabytes of data (say for calculating aggregate information), the wait can be fairly long. There are two options to remedy this situation. An easy solution is to not allow the report to be run when there are other users on the system. A second solution is to alter the default locking method. The user running the report (or the application the user is working with) can issue the following command:

```
set lockmode session readlock = nolock
```

This command instructs the data manager to run the report, and its associated retrievals, with no locks on the data. Other users are free to change data while the report is being run. The implication, of course, is that the report may access some strange data. Because the data may be inconsistent, this type of access is sometimes known as a "dirty read" of the data.

A second potential locking problem is known as deadlock, or a "deadly embrace." Deadlock occurs when one user waits on another user to release a lock and that user is waiting on the first user to also release a lock: each user is waiting for a lock to be released by the other. The typical situation that this occurs in would be two multistatement transactions. Each user issues a multistatement transaction that tries to read data, then write data (see Fig. 6-4). The data that the first user is going to read are the data that the other user is going to write, and vice versa.

Both users begin their multistatement transaction and then issue their read statements, each getting a shared lock on their respective pages. Each of these locks is granted. Then, each user issues his write request. Neither user can proceed until the other is finished. Note that this situation is very different from timeout. In the timeout situation, the report would have eventually finished running or our QBF user would have come back from lunch. In a deadlock situation, there is nothing that either user can do.

Deadlock is automatically detected by the lock manager. The lock manager picks one of the users and aborts his operation. In our example, this simply involves releasing the shared read lock on one of the pages of data. If the first operation had been a write operation, that operation would have been rolled back and the data would revert to the state it was in before the transaction began.

Next, two things happen. First, the lucky user proceeds with his transaction as if nothing happened. The unlucky user gets an error message back. One of the responsibilities of an application programmer is to decide what to do in this situation. The program could reissue the request. Alternatively, the program could issue a nasty message to the user telling him "tough luck." The more typical situation is to reissue the request.

Although a multistatement transaction is one typical cause of deadlock, lock escalation can also lead to this situation. When a query escalates to a table-level lock, page locks are kept in place until the table lock is granted. Two users can each be waiting for a table lock on the same table, and each one will not release page-level locks before the table lock is granted, leading to a deadlock situation. Lock escalation is thus bad for two reasons. First, it can easily lead to deadlock. Second, it requires a lot of unnecessary locks to be requested. Locks on a system are a limited resource and unnecessary lock requests decrease performance.

A large number of overflow pages is a frequent cause of lock escalation. The optimizer sees that only a few pages are affected by a query and takes out page locks. Then, those pages turn out to have overflow pages. Each of the overflow pages also needs to be locked, eventually resulting in lock escalation.

The third potential locking problem is livelock. In the default INGRES environment, the user waits forever for a lock. Say a user requests a shared read lock on a table, then another user wants a write lock on the same table. The write lock is placed in a queue. While the first user is reading data, another user wants to also read data.

Usually, a lock manager tries to grant as many requests as possible. In this case, the third user would be given the shared read lock, making the writer wait even more. This situation is known as livelock because it could potentially last for quite a period of time. A good lock manager detects this situation after a while and begins queuing the shared

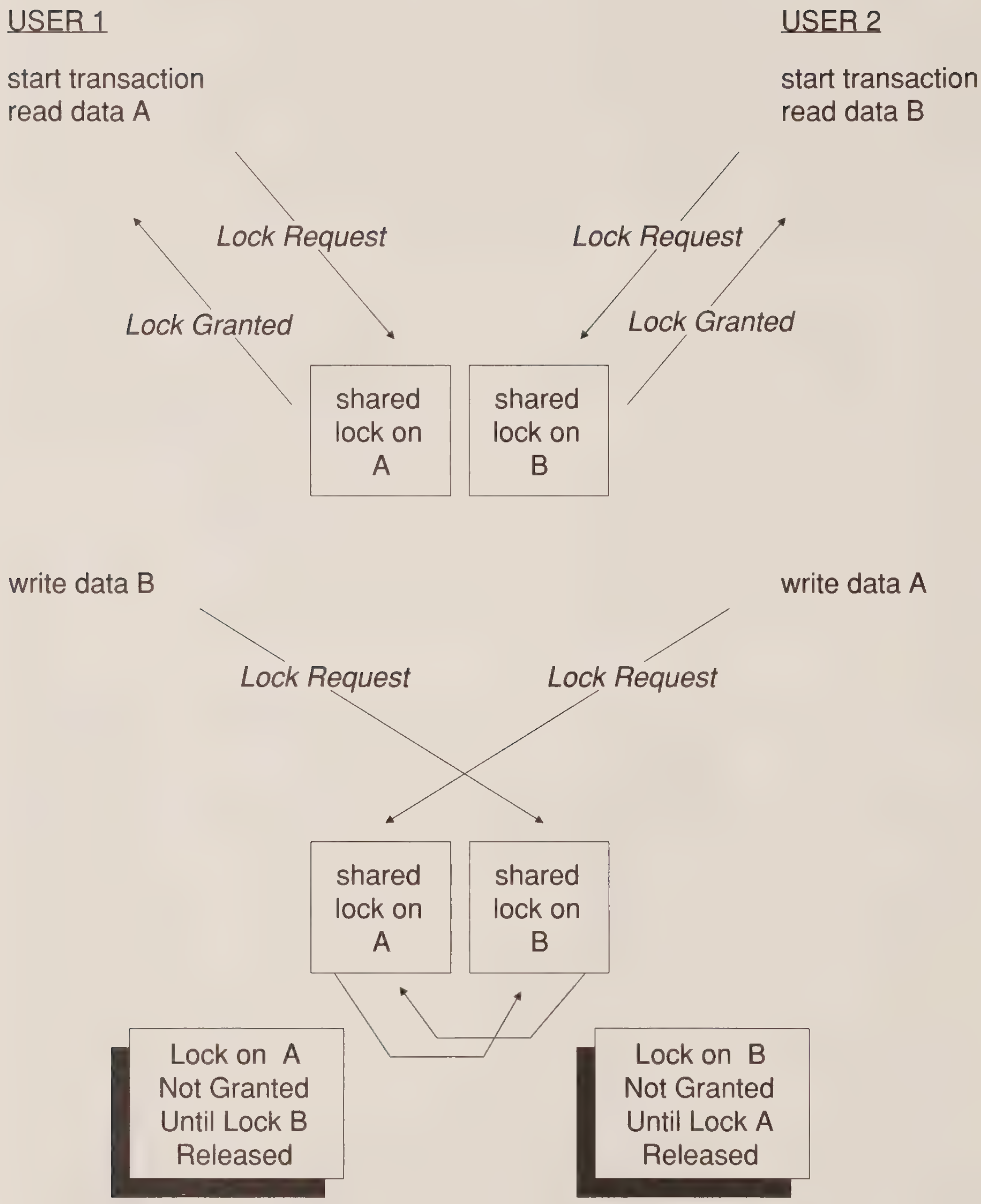


Fig. 6-4 Deadlock

locks to allow the write lock to proceed. Another possibility is to set a timeout parameter so that the write lock gives up after a while.

As can be seen, proper operation of the lock manager is essential for good performance in a multiuser environment. Deadlock is a frequent cause of poor locking behavior. The query optimizer needs to be furnished with appropriate information to make an informed decision. Using optimizedb is one way to give the query optimizer an accurate picture of the data profile so that it may make an informed decision on the proper level of locking.

It is also possible for users to modify the lock manager on a per operation or session basis. Setting a report to run with no read locks is one example of such behavior. Another frequent operation is to take an exclusive lock on data when reading it. That way, when the data are later changed, the lock does not have to be changed from a read lock to a write lock.

Changes in the behavior of the lock manager can be requested for an entire session or only for a single transaction. Users can change both the level of locking and the types of locks requested for read and write operations. Users can also set the "maxlocks" and "timeout" parameters using the "set lockmode" command. Maxlocks defines when the system escalates to a table-level lock. On a very large table, it is possible that a lot of page-level locks are desirable. Locking a 20-Gbyte table just because 11 pages of data are being changed is a fairly drastic measure. Adjusting the maximum locks parameter is an alternative to locking the entire table.

Setting the timeout parameter makes a queued lock request give up after a while instead of waiting forever. The user or application program receives an error message and any statements in a multistatement transaction are backed out. When INGRES is installed, the system manager can change various parameters for the lock manager. If the environment has high concurrency—many users—the locking tables can be made bigger. This is important because when locking resources are exhausted, the lock manager starts converting locks to table-level locks to free up space.

Another system-wide resource that affects concurrency is the data cache. When data are retrieved from disk, they are placed into a cache. INGRES consults the cache first before going to the disk drive. This is a significant increase in performance since the cache operates at main memory speeds, an order of magnitude more than a disk drive.

Each server has a cache. In a single-server installation, the cache is a global cache; the data in the cache are available for all users. In a multiserver installation, the cache is local to each server. The system manager is able to adjust several parameters on the cache manager. For example, if large amounts of main memory are available, the manager can increase the size of the cache. It is also possible to prevent specific relations from being cached.

User Control over Locking

Normally, users and application developers are unaware of the operation of the lock manager. In a few instances, programmers will try to influence the way locks are set for special-purpose applications. As a general rule, there are two extreme approaches to user-defined lock strategies:

- the brute force approach
- the delicate approach

In the brute force approach, table locks are immediately acquired to prevent the possibility of deadlock and to reduce the overhead of acquiring multiple page-level locks. This approach is used when extensive scans of data are needed, such as processing aggregate data. The approach here is to get the locks, do the processing, and go away.

The delicate approach attempts to use page-level locks whenever possible to increase concurrency, allowing multiple users to access the same table whenever possible. This approach uses several techniques to attempt to prevent escalation. First, overflow is minimized by choosing the appropriate storage structures and remodifying ISAM and hash tables.

Next, fill factors are reduced so that fewer records occupy a given page, giving more breathing room in a concurrent environment. QEPs are checked to make sure the optimizer is making informed decisions and using secondary indices whenever possible. Finally, the maxlocks parameter is set higher to prevent escalation. Often, this parameter is set to 10% of the table size, measured in pages.

When the delicate approach is used, deadlock can become a problem. On some locking systems, such as the VMS Distributed Lock Manager, it is possible to reduce the deadlock wait time from 10 seconds to some smaller number, resulting in quicker detection of deadlock. Because locking has overhead, it does not make sense to have a single user go through a lot of locks when alone on the system. This is often the case with large batch jobs that run at night. The user can specify that a lock be taken at the database level as an exclusive lock. Since the entire database is locked, write requests can proceed without first consulting the lock manager.

Another possibility is for users to do their own locking. This is done manually at the row level by setting a flag in a column and making all applications check that flag before they access data. Needless to say, this means that the front-end application has to do the locking instead of the back-end data manager. Manual locking can be used in a variety of situations. As was seen, the granularity of the INGRES locking system is at the page level. This decision to lock pages instead of rows was made as the result of several research studies at the University of California at Berkeley that determined that, as a general rule, the extra overhead in managing locks at the record- or row-level was not justified.

Users that do manual locking do so because they need a more granular locking level. For example, many records could be contained in a single page. When a write lock is taken out for updating one record, that locks all other records in that page. Manual locking allows many small records to be all updated.

There is a simple solution to this situation that does not require the application programmer to do her own locking. A page of data in INGRES is 2000 bytes (plus a 48-byte overhead). If a table is 1001 bytes wide, only a single record will fit in a page because records cannot span pages. The database designer simply adds dummy columns into the table to make the table 1001 bytes wide. Of course, this uses up more disk space; but the relevant question here is which is more expensive—another disk drive or another programmer.

Another situation requiring smaller locking granularity is where different users update different parts of the same record. In this situation, spanning a single record into a page will not work. Instead, the applications are written so that shared write locks are acquired and flags set so that different applications don't write on the same piece of data.

There is a strong disadvantage in using these techniques to do manual locking. General-purpose user interfaces cannot be used because they are not aware of this non-INGRES locking scheme. A preferable solution is to either put a single record per page or to break the table up into several relations, one for each application. Views can be defined so that the tables appear as one for retrieval purposes.

Logging and Recovery

The lock manager is used to prevent a user from performing an operation inconsistent with an outstanding operation. Locking is the way that the data manager proactively prevents inconsistency. The recovery manager is used retroactively, that is, when a transaction has aborted or a user or the system crashed.

When a transaction or other operation is started, that information is entered into a log file that contains information on all outstanding and recently completed transactions. Before any operation takes effect, it is entered into this log file. If a user enters in two statements of a multistatement transaction and then aborts and logs off the system, the effects of the two statements are indicated in the log.

The job of the recovery manager is to find unfinished transactions. When the system first initializes, the first thing the recovery manager does is look for uncompleted transactions. It takes each of these transactions and rolls it back, performing the reverse of the operation that changed the data. This recovery process is automatic; no user intervention is required. Automatic recovery is fairly simple to conceptualize when the system first initializes; it involves a scan of the log to look for all uncompleted transactions.

During a recovery procedure, the recovery manager does two other things in addition to backing out incomplete transactions. First, it goes to the system catalogs and makes sure that the physical files in the directory agree with the catalogs. This is in case a user attempted to create a new table and then aborted before the operation completed. The second thing that the recovery manager does is look for unnecessary files. If the user aborted, there could be several temporary files that were created for processing a query or sort files used by the report writer. These files are purged to free up disk space.

In order for the recovery manager to function effectively, the log file has to be intact. This is thus an essential part of an INGRES environment. The *II_LOG_FILE* logical name normally points to the same area as *II_SYSTEM*. It is important that the disk drive not fill up or the recovery manager cannot be invoked. It is also extremely important that the II_LOG_FILE disk not be I/O bound, since this is the most serious potential bottleneck in an INGRES environment. If battery backup is available for the main memory on a computer, it is possible to have II_LOG_FILE point to a RAM disk, which can greatly speed up transaction processing rates, particularly if the fast commit option (discussed later) is used with a server.

Two parameters in the recovery manager are used to adjust for different transactions rates. The number of log buffers in memory sets the number of I/O operations that can be waiting to be put in the log file. The block size for the log file I/O operations sets the size of transfer operations into the logging file.

A process related to the recovery manager is the archiver. The archiver periodically goes into the log file and removes completed transactions. This is important because otherwise the log file will fill up and no more transactions can be entered.

Two parameters set how often the archiver wakes itself up to perform functions and how much work it does each time. The recovery manager periodically writes a consistency point into the log file, telling the archiver which transactions up to that point have been successfully completed. The INGRES system manager can set a parameter to make the archiver wake up after fewer (or greater) consistency points have been written. The second parameter tells the recovery manager what percentage of the log file should be used for each consistency point. With a consistency point interval of 4, and a 5% of the log file parameter, the archiver would wake up after 20% of the log file is full.

If a transaction log file becomes full, there can be no new transactions started on the system. Two parameters set how this decision is reached: the force abort limit and the log full limit. The force abort limit is a soft failure point and is typically set to 80% of the log file's capacity. When this soft failure point is reached, the oldest pending transaction is aborted. Hopefully, this keeps the log file from getting too much bigger. Note that the remaining active transactions could write a great deal of data into the log file, causing it to still increase in size.

The log full limit parameter is reached typically at 95% of log file capacity. This is considered a hard failure. At that point, all new transactions are stopped until enough space is freed up by aborting the oldest transaction still outstanding. If the log full limit parameter is exceeded, the system manager should do two things. First, the nature of the transactions should be examined to see if this was a typical situation. If a department started its Christmas office party with a lot of unfinished transactions still active, and another department continued doing work, that might explain the reason for the problem. If the situation is expected to reoccur, or might reoccur under unusual circumstances, the size of the log file should be increased. Increasing the log file allows more uncompleted transactions to remain active.

Checkpoints, Journals, and Audits

When the archiver removes data from the log file, it has two choices of what to do with the data. First, it can simply discard the information. Second, it can move the information into a journal file, which is a more permanent log of transactions. Related to a journal file is a checkpoint of the database. A checkpoint is a snapshot of the database at some point in time. The journal is a list of all transactions since that checkpoint. The recovery manager is able to handle the problem of an inconsistent transaction that was aborted, either due to a front- or back-end abort or a system crash. Checkpoints and journals are used to handle more permanent problems with data.

An obvious use for a checkpoint is when the disk drive containing the data is damaged. Assuming the checkpoint was kept on a different disk drive, this provides a snapshot of the data in the database. The *rollforwarddb* command is then used to apply journals to the checkpoint to get the database back up to date. Any transactions that have not been archived are lost, but everything else is recovered.

The rollforwarddb command is also quite useful for another type of situation—human error. Let us say we take a checkpoint of our database and keep a journal file around. At some point, a new programmer goes in and performs the following SQL statement:

```
update emp e set salary = 10000 where staff.name = "Martin"
```

The intended purpose of this query was to reset the salary on one row. Our programmer forgot to join the emp and the staff tables together so that only the row that matched got changed. Instead, the data manager will form a cartesian product of the two tables—every possible combination of rows in the two tables. For every row in the emp table there will be at least one match with a row of the staff table that has *name = "Martin"*. This is known as a disjoint query and results in every single row being changed. When the change is detected, the checkpoint and the journal can be used to bring the database back to the point just before this potential disaster occurred.

A checkpoint automatically makes a snapshot of every table in the entire database. It is not always desirable to activate journaling for every table, however, because updating journals requires processing power and takes up disk space. When creating a table, the user is able to specify *with journaling* to indicate that this particular table is to be logged in the journal files. If a table is not journaled, then the rollforwarddb utility will not be able to update the table with transactions that occurred after the last checkpoint.

The *auditdb* command is used to examine the journals. The user can perform an audit on the database, specifying beginning and ending dates, tables required, or possibly all databases owned by a particular user. Usually, this is then turned into a file in INGRES bulk copy format. The user would then copy that information into a database, first creating the appropriate table:

```
create auditrel_table (      date = date, user = char(24),
                             operation = char(8), tranid1 = integer,
                             tranid2 = integer, tbl_id_base = integer,
                             tbl_id_index = integer,
                             {columns of table being audited }  )
```

Auditdb is a valuable tool for security and accounting purposes. For security purposes, the audit trail allows the analyst to trace which operations occurred on which tables in the database.

The accounting features of auditdb are also extremely useful. On most computer systems, the operating system accounting utility operates at a very gross level. Usually, the utility will keep track of which programs were executed, and how much computer resources the program used. If all people on a system are using INGRES, the operating system accounting utility will not necessarily provide enough information for either chargeback or planning purposes. Coupling the auditdb utility with the operating system accounting package, however, can provide a great deal of information.

As an example, think of a prototyping environment. Several users are on the system testing applications. At some point, the applications will be expanded to include the entire organization. In order to expand the application user community, there will probably be an acquisition of new computer equipment. The question is, how much equipment to buy?

With auditdb, it is possible to keep track of when particular operations were performed on the system. The accounting package on the operating system can keep track of gross usage on the system. Finally, a monitoring utility can show the response of the computer at any point in time in terms of idle CPU cycles, disk drive saturation, and other components.

All three types of information can be loaded into a database and used for planning. Reports can be run that show the amount of resources used by certain operations and the performance of the system. Based on these reports, a quantitative estimate of the impact of expanding the user community can be reached and an informed decision on the amount of computer resources needed to support the applications can be made.

Increasing Performance

At this point, we have examined the five basic functions that are present in a single-system version of INGRES (see Fig. 6-5):

- a front-end application
- the data server
- the lock manager
- the recovery manager
- the archiver

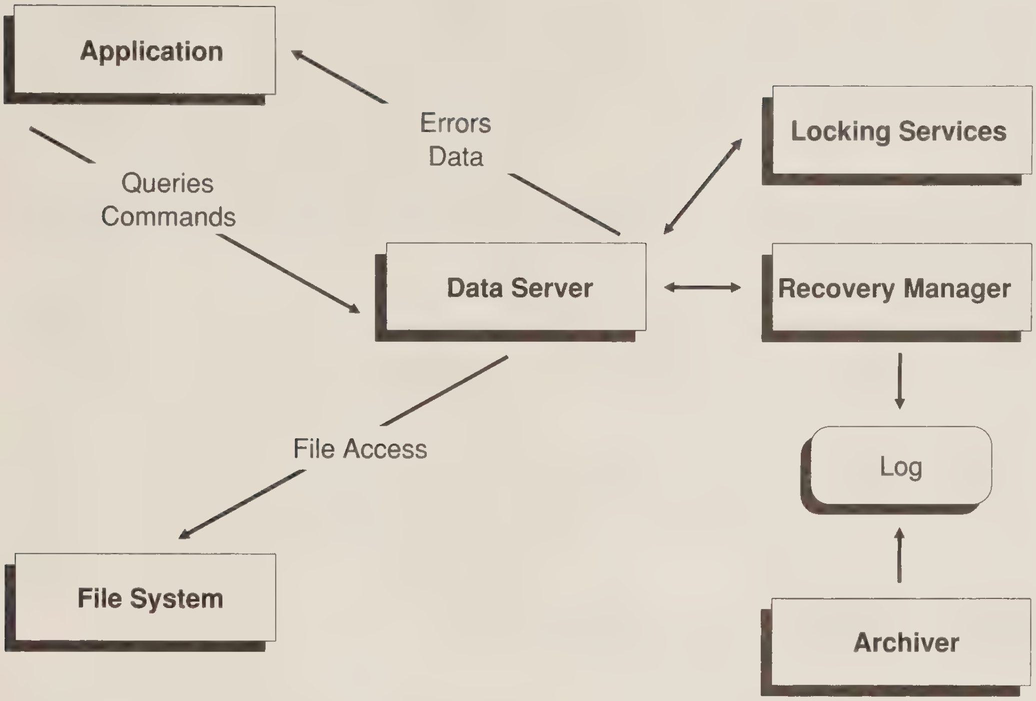


Fig. 6-5 Components of an INGRES Implementation

Each of these components, when properly configured, helps provide fast access to information while protecting the integrity and consistency of data. Because of the power available with these tools, however, it is also possible to configure them in a way that has the reverse effect.

A typical scenario for a new INGRES installation consists of the following. INGRES is brought in and installed, and a test application is written. The application is run, and performance on the user's computer degrades tremendously. Not only does the new application run slow, but everybody else's applications also run slow. The typical reaction to this scenario is to tune the system. Most users have an intuitive understanding of the fact that a report on a database of moderate size (say 10 to 20 Mbytes) should not take 10 CPU hours to run. A performance analyst can often make that same report run in a matter of seconds. Often the performance analyst is instructed to begin by tuning the system, say a VAX. This involves tuning VMS sysgen parameters and other VMS system-level functions. However, it is a very rare system that is so badly tuned that an analyst is able to get a 600-fold increase in performance!

So how does the analyst make the report run in under a minute? The answer is almost always in the design of the application and the database, not in the tuning of the operating system. Think about a poorly formed query that joins two tables. If each table has 10,000 rows and the query results in a cartesian product, the report needs to do a minimum of 100 million I/O requests. If the data were to be sorted, the number of I/O operations can increase again by an order of magnitude.

If our user only wants to join 100 rows out of that database, it is obvious that a proper primary or secondary index can reduce the number of I/O operations from 100 million to something like 1000 or less. The moral to this is that any performance analysis needs to begin by looking at the database design and at the SQL queries that form the basis of the application. Only then should attention shift to broader issues like operating system parameters.

To increase performance, it is first important to understand the nature of the application. Applications are complex, as are the underlying computer systems. It is thus impossible to come up with rules of thumb that work in all situations. This section describes three ways in which INGRES can help increase performance in a multiuser environment:

- functions inherent in the INGRES database
- ways of tuning the operating system to increase INGRES performance
- a diagnostic checklist for application and database design.

INGRES Features to Increase Performance

Three features in INGRES allow the data manager to achieve very high transaction processing rates. These features are:

- group commit
- fast commit
- the multi-server architecture

Normal INGRES execution requires each transaction to be written to the log file on disk. This way, if there is a system crash, the definition of the transaction is on disk and the recovery manager is able to roll forward the database to ensure consistency. The problem with this is that a single drive's I/O rate is usually limited. Even though the CPU can accept large amounts of transactions, it will be limited by its ability to write the transactions to the disk drive.

The group commit capability allows several transactions to be grouped and committed at one time. This piggybacking ability means that a single I/O operation can be used to commit several transactions, greatly increasing the transaction processing rate on a system.

Related to the ability to group transactions in a single I/O operation is the fast commit facility. Normally, both the data pages and the log file must be updated on stable storage. The fast commit capability allows only the log file to be written to stable storage for a transaction to be committed. In the case of failure, the recovery manager is able to roll forward the database to add the transactions that may not have data pages written to disk.

Perhaps the most significant feature of the INGRES environment is the ability to take advantage of multiprocessor computers. Since multiple data managers can work with the same database files, each of the data managers can be put onto a different processor. Two examples of these high-performance environments are the Sequent parallel processor and VAX Clusters.

A VAX cluster consists of several different VAX computers connected together in a high-speed network. This network uses the CI (computer interconnect) bus to move data at 70 million bits per second. By contrast, the Ethernet local area network operates at only 10 million bits per second. Also connected to this high-speed network are dedicated disk controllers, known as a Hierarchical Storage Controllers (HSCs). An HSC can have up to 32 disk drives attached to it. The purpose of the cluster is to allow multiple computers to access the HSC.

Coordination of access to the HSC is done by the VMS distributed lock manager, which allows the different VAX computers to find the manager for a disk resource such as a file and obtain a lock on that resource. If one of the systems crashes, the cluster control software is able to rebuild the lock space to ensure that the integrity of the data is maintained and that processing can continue.

With a VAX cluster, up to 24 VAXs or HSC controllers can participate in this high-speed network. Disk drives can be dual-ported, that is, connected to two different HSC controllers, for redundancy. Data can also be shadowed—copies of the data can be written on two different disk drives in case of a disk drive failure. To the user of the cluster, this environment appears as a single computer. All of the clustered computers have access to the same data. Terminal servers are used to log the user onto the VAX that has the best performance at the time.

The advantages of the cluster are threefold. First, there is high availability in case of a failure in any of the components. Second, there is a modular upgrade path. Disk drives, controllers, and computers can all be upgraded without changing the rest of the configuration or changing the appearance of the system from the point of view of the user. Most importantly, the cluster offers extremely high processing capabilities. The HSC controllers assure adequate bandwidth for data access, often a bottleneck in high-performance configurations. Multiple VAXs can all process the same user information.

INGRES runs very well in this environment due to the multi-server architecture. Each of the VAX computers can have one or more servers on it. The VMS distributed lock manager is used to coordinate the access of these different servers to the underlying data on the HSC controllers. Normally, there is only a single logging process and log file on an INGRES system. In a clustered environment, there is a separate log file for each of the back-end nodes in the cluster. A DMF cluster server process (DMFCSP) runs on each node of the cluster to maintain transaction consistency across systems.

When there is a failure in a clustered environment, all of the DMFCSP processes communicate. A single node is designated as the master recovery node and acquires the failed node's recovery lock. The master node then backs out all aborted or in-progress transactions, then allows the various servers to continue processing.

It is possible to have local disk drives in a clustered environment. These disks are not available to the rest of the nodes in the cluster. It is important that the log files be kept on a cluster-available disk drive because otherwise the designated recovery node will be unable to perform its function.

System Tuning

A VAX, Sequent, Sun, or any of the other processors that INGRES runs on is a general-purpose computer system. The same computer can run electronic mail, word processing, or a variety of other applications. Each of these systems has different operating characteristics. Electronic mail, for example, consists of users who do a lot of typing of small files. The memory requirements are fairly small and the amount of I/O is low. A database operates in sharp contrast to the electronic mail operating characteristics. When a user buys a VAX with the VMS operating system, it is usually tuned for electronic mail. Consequently, the database may run slower than it could.

Operating systems such as VMS allow the manager to set a variety of characteristics. One characteristic is how much memory a user gets when logging onto the system. Relational Technology recommends that this parameter, known as a working set, be increased from the default value for database users. The working set parameter is an example of a user parameter. Other user parameters include the amount of disk space or the number of CPU cycles the user is entitled to. It is possible to set these parameters on a per user basis, for batch queues, or for an individual program. Often, the batch queue will be set to run at lower priorities than the interactive users.

There are also system-wide parameters. One such parameter on VMS sets how often the system stops to evaluate memory utilization. If an individual process does not have enough memory, it is forced to swap lesser-used data. This slows down system execution because the system is continually moving data back and forth from the disk drive instead of spending CPU cycles doing the actual processing for the user.

Relational Technology lists a large number of suggested parameter changes for tuning the various operating systems that INGRES can run on. These parameters may be different from the default settings and can help optimize performance for the particular characteristics of the data manager and the user interface processes.

Because different programs have different operating characteristics, there is only so much tuning that can be done in a multiprogram computer. A distributed network allows different computers to be dedicated, and hence optimized, for particular tasks. The user interface can run on one computer, the data manager on a second, and electronic mail on a third. The next part of this book examines the General Communication Facility (GCF), which allows different data repositories to be distributed throughout a heterogeneous network in a transparent fashion for users.

Performance Checklist

As discussed earlier, bad database and query design can lead to cartesian products and other anomalies that greatly increase the amount of I/O on a system. When examining poor performance, it is important to first look at these types of issues, then look at lower level functions such as the system configuration. The following is a recommended diagnostic checklist. The first items should be examined first:

- database design
- structures and indices
- key types (multicolumn, sequential, high duplicity)
- validation checks and integrity
- permits and views
- Embedded 4GL or INGRES 4GL code
- concurrency
- operating system
- hardware

Of the topics in this checklist, database design has not yet been considered. That topic will be discussed in Part IV.

Storage structures and indices are one of the first items to consider in examining poor performance. When the query optimizer is unable to directly access data, a scan of the entire table is performed. For large tables, especially for joins between large tables, the amount of I/O can be quite significant.

If the tables seem to have the proper storage structures and secondary indices, it is possible that the key values that were chosen are not appropriate. A multicolumn key, for example, can lead to a wide index, reducing the effect of the index. Sequential keys can easily lead to a high number of overflow pages on a static index like the ISAM storage structure. High numbers of duplicate rows can also lead to large numbers of overflow pages.

Validation checks, integrities, and security constraints and views can all make a seemingly simple query quite complex. Permits, integrity constraints, and views all modify the query, tacking on additional clauses or expanding the definition of a table from the underlying view definition. Validation checks in the front-end application can also have a similar effect. For example, in VIFRED it is possible to specify a validation check that says that the value on a field must be contained in an existing column in a database table. At run time, the values currently in the referenced column are all loaded into memory for the user. If several users all have the same application, it is possible that a very large amount of memory is being used, reducing performance.

Another possible cause of poor performance is poor application design. Inefficient use of the INGRES 4GL or a third-generation language can lead to infinite loops and other contributors to reduced response time. After all these factors have been examined, it is time to begin looking at lower-level issues. If an application does not run effectively in a single-user environment, it does not make much sense to look at locking as the first indicator of poor performance.

On the other hand, an application may run quite efficiently as a single-user application, but operate slowly in a multiuser environment. The performance analyst can examine the transactions in the application and monitor the QEPs to look for deadlock, livelock, lock escalation, and other concurrency problems. Some operating systems have utilities to examine the numbers of locks queued at a particular point in time as an aid to determining the level of concurrency.

The last set of issues to examine are the operating system tuning parameters. Two problems are frequently found. First, many memory managers on an operating system are tuned for many small applications that do not use a significant amount of main memory. The memory manager can be adjusted to take into account the memory-intensive nature of database systems. The second typical problem is disk saturation. If all of the INGRES files are contained on a single disk drive, this component may be a bottleneck and cause poor performance. Moving key components, such as the log file, to other disk drives can significantly help performance in this case.

Summary

The last chapter discussed storage structures, indices, and the query optimizer. This chapter moved these single-user performance issues into a multiuser environment, including the configuration of INGRES on a particular operating system and the functioning of the lock manager. The computer consists of main memory and disk drives in addition to the CPU. By properly distributing data and other files onto different disk drives, the performance of the data manager is optimized.

The data server is the program used to access these files. It is able to service multiple users, thus conserving memory. Other portions of memory are used as cache space so that the server is able to access data at main memory speeds instead of accessing the disk drive.

The lock manager is used to coordinate access to data in both single-server and multi-server environments. Locks can be exclusive or shared and can operate at a variety of different levels. A shared lock on the database is used to ensure that no write operations begin while read operations are active. An exclusive lock on the database is used to keep other users out and reduce locking overhead for large jobs.

Page- and table-level locks are used for normal data access. The query optimizer decides at what level to take a lock depending on the amount of data it estimates will be found. If page locks are taken, and more data are found than expected, then the lock escalates to a table-level lock.

Lock escalation and multistatement transactions are both causes of deadlock, also known as a deadly embrace. Deadlock is automatically broken by the operating system and one of the users is allowed to proceed. Another locking problem is timeout. When a user runs a very long report, she can tie up lock resources for a long period of time. Normally, the other users will wait until the locks are available. In the case of a large report running in batch mode at low priority, this can be a considerable period of time.

Low-priority reports are examples where users might manually set the locking strategy. Normally, this is done transparently to the users. When the default locking strategy is not optimal, the user can alter that strategy. In this case, the user might set the report to run without any locks instead of a shared read lock.

The recovery manager and archiver are used to ensure that transactions are properly processed on a system. The recovery manager looks for aborted transactions and automatically restores the database to a consistent state. The archiver periodically cleans completed transactions out of the log file and moves them to a permanent journal. The journal and checkpoints are used to recover large amounts of data in the case of data being destroyed or damaged. This might be because a disk drive crashed, or because a user performed an undesirable operation.

Finally, a variety of enhancements to performance were discussed for providing high transaction processing rates. The group commit and fast commit facilities allow transactions to be quickly committed, thus bypassing usual bottlenecks in a database environment. The multi-server architecture allows multiple processors on a parallel processor to have data servers and coordinates the actions of these multiple servers.

Chapter

7

Extending the Data Manager

Overview of Postgres

Postgres is a research project at the University of California at Berkeley, under the direction of Professors Michael Stonebraker and Lawrence A. Rowe. Both professors are founders of Relational Technology and are still active in setting the overall direction of the company. Postgres, however, is not a commercial system. It is a research project carried out in the public domain.

In the past, many of the features in INGRES have come from research environments. The original data manager, for example, is an offshoot of University INGRES, one of the first relational database managers. ABF is an offshoot of a research project developed by Professor Rowe called the Forms Application Development System (FADS).

Postgres represents a fairly radical enhancement to present commercial database systems. It is discussed for two reasons. First, because of the players involved, it is likely that some of the features will eventually find their way into INGRES. Which features will be incorporated is very much a marketing decision by Relational Technology. A more important reason for discussing Postgres in this book is because database management systems are dynamic, continually changing, expanding features, and increasing performance. Postgres gives us a glimpse of where these systems are going. It is an imperfect glimpse of what tomorrow's commercial systems will look like, but a glimpse nevertheless.

Postgres attempts to solve several problems with today's commercial systems. Although many different research goals are present in the project, and these goals change over time, a few of the goals are:

- the ability to deal with complex objects through simple queries
- the ability to extend the database with new data types, new access methods, and new operators
- an active database that is able to respond to changing data through alerters and triggers
- support for high availability and high performance on modern hardware architectures

To begin a discussion of how databases will change in the future, we must first look at how the underlying hardware platforms will change. Postgres is not built to run on today's systems—a PC version of Postgres is a contradiction in terms. Postgres is instead developed with tomorrow's systems in mind.

First, systems will be increasingly powerful. Bill Joy, Vice-President of Sun Microsystems, has developed a widely quoted approximation of the power of single-chip CPUs known as Joy's law:

$$\text{MIPS} = 2^{\text{(year - 1984)}}$$

In 1988, many single-CPU computers have 16 MIPS. In 1989, Joy's law predicts single-CPU computers of 32 MIPS; and in 1990 and 1991 we are looking at computers with 64 to 128 MIPS becoming commonplace. Postgres is built to take advantage of these increasingly powerful CPUs.

Not only are individual computers becoming increasingly powerful, but parallel processors help augment the powers of these individual systems. The 32-MIP computer is really a very small data repository or a user workstation—a personal computer, if you will. Parallel processors yield computers with several hundred MIPS. Postgres is designed to take advantage of the presence of parallel processors. For example, the query optimizer is able to break an individual query plan down into several fragments, each one being executed on a different processor. These assumptions about the power of computers are not just futuristic visions—these computers exist. Many large-scale environments are already using very powerful computers. For example, Sequent markets a parallel processing system that is frequently used for INGRES applications.

The CPU by itself is not especially useful without some links to the outside world. Mass-storage peripherals are also keeping pace with the increase in CPU power. Postgres can effectively use systems with 0.25 to 1 Gbyte of main memory, large arrays of magnetic disk, and very large amounts of tertiary storage in the form of optical disk jukeboxes or magnetic tape.

Postgres builds on this underlying hardware platform to add the next level—the data access mechanism. In traditional computers, this is known as a file system. On a VAX, for example, most data access is done through the Record Management Services (RMS). The assumption of many operating system and hardware developers is that relational DBMS will replace traditional file systems. All access to data will be through the database.

This assumption is also not far-fetched. SQL-based data access is an integral part of IBM's System Application Architecture. DEC is also beginning to incorporate Rdb heavily into system functions. For example, DEC network management tools are based on Rdb instead of RMS files. One of the keys to using a DBMS for all data access is the ability of the data manager to handle the complexity of data types found in traditional file systems.

Object Management

Postgres extends the QUEL query language to be used for the management of complex objects. POSTQUEL consists of the basic QUEL syntax extended in a variety of ways to support the management of these complex objects through simple queries. Like all database systems, Postgres supports a variety of native, simple data types. These are similar to the INGRES data types and include integers, floating point numbers, and characters. Most applications, however, consist of more complex forms of information.

When the database does not support these more complex forms of information, such as dates or arrays of characters (a text string), the programmer is forced to resort to encoding information. Date, for example, can be represented as a column of type text in the database. In order to answer a query requesting all events before a certain date, the programmer is forced to develop an algorithm that interprets the text string and decides if each one falls before the target date.

Abstract data types allow the data server to answer these questions in the back end, freeing the programmer to concentrate on other issues. Date is an example of an abstract data type. Abstract data types are one of several extensions to the native data types available in Postgres. This section will discuss three types of extensions:

- arrays of simple data types
- complex data types (a collection of several data types represented as a single object)
- abstract data types (new types of data added by users of the system)

In addition, this section will discuss several extensions to the query language that are particularly useful for the management of objects. Inheritance allows objects to be structured in a hierarchy, such as the shared object hierarchy used in the Picasso project. Transitive closure allows the programmer to have a query performed in an iterative fashion until no more results are found.

Arrays of Simple Data Types

The SQL VARCHAR is an example of an array of simple data types. VARCHAR consist of an array of characters. In a language such as SQL, characters are the only type of data that can be structured into an array. Postgres allows any simple data type, such as integer, floating point, text, or date, to be structured in an array. The array can be of fixed size or of variable length. Thus, a programmer could create a relation that contains an array of dates:

```
create emp ( name = char[20],date_paid=date[] )
```

To retrieve information from these arrays, the query can omit the array subscript designator and thus retrieve all the information. For example, the following query can be submitted:

```
retrieve ( E.name, E.date_paid)
        from E in emp
```

This query retrieves all characters of the name, and the entire array of dates paid.

Alternatively, the user can retrieve certain portions of the array. For example, to retrieve the first 10 letters of the last name and the first date, the user would submit the following query:

```
retrieve (E.name[1:10],E.date_paid[1])
        from E in emp
```

Arrays of simple data types are extremely useful in a variety of situations. Text, an array of characters, is an obvious application. Arrays of numerical data types are useful in financial applications that deal with series of numbers. Finally, arrays of graphic data types are useful in applications such as CAD/CAM that build primitive graphic objects into more complex objects.

Complex Data Types

A complex data type allows a user to refer to a piece of information as a single object. That object can in turn be composed of several pieces of information. There are two forms of complex data types in Postgres:

- attributes of type relation
- POSTQUEL procedures

An attribute of type relation allows a user to define a new data type that is in fact a retrieval from an existing relation. Whenever a relation is defined to the system, that relation becomes a new data type. A user can define an employee relation, for example. Then, a new relation, department, can be constructed as follows:

```
create dept (name =char16, budget=int4, mgr=employee)
```

A special form in the query language, known as nested dot notation, allows the user to retrieve one of the components of a complex data type. For example, a user may wish to see the name of an employee and the name of the employee's manager. To do so, the user would submit the following query:

```
retrieve ( D.name, D.mgr.name )
        from D in dept
```

Notice that the retrieval now has three levels that specify the particular piece of data to be retrieved: the name of the relation, the name of the attribute, and the name of the subattribute within the complex object. A complex object can itself be made up of complex objects. The nested dot notation can be nested to arbitrary levels to reference the components of an object.

This can also be used any time a column is referenced, such as qualifying a query. To retrieve only employees that work for a particular manager, the user can enter the following query:

```
retrieve (D.name )
        from D in dept
        where D.mgr.name="Smith"
```

In addition to attributes of type relation, Postgres supports procedures as another form of complex data type. A procedure is a collection of POSTQUEL statements that is executed whenever the column is accessed. The attribute of type relation is actually a special form of procedure; whenever the column is accessed, a retrieve statement is constructed by Postgres.

With INGRES, procedures are all stored in one table. The procedure is then only available with an execute statement, which goes to the special procedure table and executes it. In Postgres, the procedure is part of any relation. The user can thus define a column in the emp table that contains a procedure for each user that calculates his performance:

```
create emp (name=char[16], perform = pquel )
```

The perform column is of type pquel (POSTQUEL). Each row of this table will contain both the name of an employee and a procedure for calculating the performance of the employee. Note that each row in the perform column can have a different procedure. It is possible for each procedure to return different types of data. The procedure for a salesperson might return the columns sales and quota, while the procedure for a manager might return the columns employee_turnover, budget, and expenses.

The nested dot notation can also be used with these constructs. To retrieve all employees that have a low employee turnover, the following query would be submitted:

```
retrieve (E.name)
      from E in emp
      where E.perform.emp_turn  < .1
```

Only those rows that have the emp_turn column are considered. The data are then further restricted to return only those rows that have a value less than 10%.

Another type of procedure is the special POSTQUEL type. A column of type pquel has a different procedure in each row, while the spquel data type uses the same procedure in each row. The special procedure is first defined as a data type, and then a table is created with a column of that data type. For example, a sales table could contain the name of the salesperson as well as the total sales for that person. First, we would define the new type tot_sales:

```
define type tot_sales as
      retrieve ( total = sum { m.sales
               from m in monthly_sales
               where m.name = $.name } )
```

The string $.name refers to the name column in the relation that contains a column of this procedural data type. The "$" in $.name is an example of a parameter. In this case, the $ stands for the name of the relation containing the data type.

The sales table can then be defined as follows:

```
create table sales ( name = char[16], total = tot_sales )
```

Every time that the column sales.total is accessed through a query, the name in the current row is substituted and the value for the procedure is derived.

A further example of a parameter is as follows:

```
define type tot_sales as
      retrieve ( total = sum { m.sales
               from m in monthly_sales
               where     m.name = $.name and
                         m.product = $1 } )
```

Here, “$1” stands for the first parameter that is passed into the procedure when it is executed. When a new row is added to the emp table, it would be added as follows:

```
append sales ( name = "joe" , tot_sales = "screwdriver" )
```

The value “screwdriver” would be substituted for the parameter $1 and the summation of sales would be restricted to screwdriver sales for “joe.”

To increase efficiency, procedures can be precompiled and precomputed. A precompiled query means that a QEP is kept in addition to the POSTQUEL commands that make up the procedure. If the definition of the procedure changes, the precompiled version of the query is invalidated and a new one is generated. A precomputed result keeps the results of the procedure as well as the POSTQUEL commands and the query execution plan. A precomputed result is invalidated when any of the parameters or values used by the procedure changed.

When Postgres has idle time, it uses that time to look for procedures that can be precompiled or precomputed. This means that when the query comes in from a user, the results may be already waiting.

Functions and Aggregates

To supplement the POSTQUEL language, users are able to write their own functions. An example of a function in INGRES is the sine function, which returns the sine of a parameter. The user could have calculated the result using a programming language or by constructing the proper equation in SQL. It is more convenient, however, to select the sin(a.angle) than to repeat the formula each time a sine is needed.

With POSTQUEL, users can write their own functions in a programming language and access them from the query language. For example, a user may wish to write a small program in C that accepted a date as input and returned the day of the week that the date falls on. The query would look like this:

```
retrieve (E.name )
        from E in emp
        where day_of_week (E.hired_date) = "Tuesday"
```

In order for Postgres to know about this day_of_week function, it must be registered. The user tells Postgres about the location of both the source code and the compiled version of the program, as well as the language that it is written in. The user also tells Postgres what arguments the function can accept and the return type. For our day_of_week example, the following command could be used to register the function:

```
define function day_of_week
        ( language = C,
        file = "/postgres/user_extend/src/day_of_week.c",
        returntype = char[16] )
        arg is ( date )
```

Functions can be included in any POSTQUEL command, as well as in procedures. A special flag on the function, “iscachable,” tells Postgres that the value of the function can be precomputed. In the case of the day_of_week function, precomputation is no problem. On the other hand, if the function returned the current time of day, precomputation might not be such a good idea.

Functions can be designated by the database administrator as untrusted or trusted. A trusted function runs within the same memory space as the Postgres code. Running as part of the same process as Postgres means that the overhead of interprocess communication is saved, and performance increases. On the other hand, a trusted function has access to all the internal data structures used by Postgres, and could, if poorly written, corrupt some of them.

An untrusted function runs as a separate process on the same computer as Postgres. In INGRES, the front-end application and the data server are examples of separate processes. Different processes use a form of interprocess communication (IPC) to communicate, whereas functions within the same process can directly access the piece of memory they wish to reference.

A function usually operates on a single piece of data. An aggregate is a special type of function that operates over several rows of data. Sum, for example, is an aggregate that adds up numbers in several rows of a table. To define a new aggregate, the user has to supply two functions. The state transition function is executed once for each row of data; it receives a state from the query executor, applies the new piece of data to it, and returns the new value of the state. The other function is a final calculation function. The final calculation function takes as input the final internal state and yields a result.

To compute an average, the state transition function would add the new data item to a running total. The final calculation function would take the final total and divide it by the number of items involved. Thus, to register the aggregate avg, the user would submit the following:

```
define aggregate avg as (
        add_new_value_function,
        divide_by_total_function )
```

Aggregates form an extension of POSTQUEL language. The aggregate can compute a single value, such as the sum of all sales in a table. Alternatively, the aggregate can compute a set of values. An example would be computing the sum of sales by department.

Abstract Data Types: Extending Postgres

Complex objects are a collection of simple data types. Postgres allows the user to extend the set of simple data types available. For example, the user might have a CAD/CAM application. The database would thus consist of a series of geometric objects. A turbine, for example, can be expressed as a complex object that is in turn made up of a series of lines, circles, and boxes. In most database systems, the concept of a box is expressed through the mechanism of encoding. All the users of the database agree that a box is represented by, for example, the coordinates of two of the points. This information might be stored as a text string.

The problem with this approach is that the data have to be decoded to be used. In addition, the query language cannot easily be used to perform operations on boxes. Not only must the information be decoded, but the programmer must then code in operations, such as comparing the area of two boxes. An abstract data type thus consists of a representation of an object (box) and various operations on that data type.

Postgres allows the user to define both data types and operators on those data types. An operator on box might be area=. The user could then submit the following queries:

```
create box_list ( box_id = int , box_list = box )
retrieve ( b.box_id)
        from b in box_list
        where b.box_list area= 24
```

Here, the user created a relation called box_list, then retrieved all boxes with an area equal to 24. Other operators, such as area less than or area greater than, can also be defined.

The definition of a new type of data is conceptually very easy. The user supplies two functions that transform the data. The first, the input function, takes an external representation of an object and transforms it into the internal representation that will be stored in the database. In the case of a box, we can store the list of coordinates that defines the box as an integer array. A user would want a more user-friendly representation, such as *20,20:30,30*. The first function would thus transform the user-friendly representation into the integer array. The second function performs the reverse transformation.

Another example would be the date abstract data type. The external format of a date might be something like "Month-Day-Year." In the case of INGRES, there are a variety of different legal external formats. The internal format, on the other hand, would consist of the number of seconds since some arbitrary starting point (i.e., Jan 1, 1900).

Operators are defined by a set of functions. At the basic level, the user supplies a single function:

```
define operator area_equal ( procedure = area_equal_proc )

define operator not_area_equal ( procedure = not_area_equal_proc)
```

Usually, operators are defined as a family of operators. For example, the operator area_equal is the reverse (commutator) of the operator not_area_equal. Less than is the commutator operator for greater than. The user who defines operators can specify the commutator operator, which allows the query optimizer to use whichever operator is appropriate. The user can also specify a sort operator that is associated with the area_equal operator. Less than is an example of a sort operator for equals. Area less than would be a sort operator for area equals. By specifying a sort operator, the query executor is able to sort the values in two columns being joined instead of performing a cartesian product.

Access Methods

Related to new operators and data types are new access methods. The basic access method, used to store all relations in Postgres, is the heap data structure. New access methods, such as a BTREE, are used to provide secondary indices to the base data. To define a new access method to Postgres, the user defines a series of low-level functions that are called by the query executor. These functions are somewhat equivalent to the functions of the INGRES DMF. For example, one of the functions is used to begin a scan. This function accepts a scan key, such as *box_area less_than constant*. Another function is used to get the rows that satisfy the scan query.

The responsibility of the begin scan function for a particular access method is to take a scan key and be prepared to retrieve all index tuples that satisfy the scan criteria. The get next function is responsible for returning the tuple ID in the base table that meets the scan criteria. In addition, the access method designer can provide functions that give information to the query optimizer. For example, given a particular scan key, a function can return the number of pages or tuples that would be returned. The query optimizer can use this information to choose the best access plan.

An index for a table consists of a key column that is sorted and the associated tuple in the primary table for each of the key values. With Postgres, sorting data is not necessarily straightforward. A box, for example, could be sorted by the area of the box, the perimeter of the box, or a variety of other operators. When a user creates an index, she specifies not only the column to be indexed but which operator class to use. Each operator that can be used with an access method is a member of an operator class, one of which is the sort operator for the class. For example, the area operators for boxes would use area less than or equal to as the sort operator.

A BTREE index on box using the area operator class would sort the data based on the area of the box. The index could then be used in any query that retrieved data with a qualification that restricted data based on box areas.

Designing new access methods is a fairly complex endeavor. The Postgres designers anticipate that only a few users will design new access methods, and that these access methods will then be made available to other Postgres users.

Inheritance

Inheritance allows a new relation to inherit tuples and columns from an existing relation. This Postgres feature lets information be placed in a hierarchy, such as the Picasso shared object hierarchy discussed earlier. An example of a hierarchy might be a personnel database. People have certain characteristics, such as sex and age. Employees have all those characteristics, plus several others such as job titles. Hourly employees have additional characteristics, such as an hourly rate.

The following data definition statements could be used to implement a personnel hierarchy:

```
create people ( name = char[16], sex = char, age = int )
create employee ( job_title = char[16] )
        inherits ( people)
create hourly ( rate = money )
        inherits ( employee )
```

When a user retrieves data from the people table, the column's name, sex, and age are returned. When the user retrieves data from the employee table, the additional column job_title is also returned. Likewise, the hourly table will return an hourly rate in addition to job_title and the information in the people table.

Normally, a query against the people table will only return the rows of data that are directly in the people table, and not from those who inherit from it. A special construct of the query language allows the user to retrieve all rows that are in the people table as well as any relations that inherit that table:

```
retrieve ( people*.all )
```

Adding a * to a table name signifies that all tables inheriting the named table should also be included in the query.

Inheritance is especially useful in constructing a class hierarchy. In Picasso, for example, there are classes of objects, such as different kinds of fields. All fields have certain attributes in common. Special forms, such as a table field, have an additional set of attributes. Inheritance allows this hierarchy to be stored efficiently in the database.

Transitive Closure

Transitive closure is a query construct that allows a query to keep running until no more data are affected. For example, the emp table might include the name of an employee and the manager. Using POSTQUEL, the following query can be run:

```
retrieve ( EMP.all )
```

The POSTQUEL query retrieves all employees and the names of their manager. A more complex query would be to retrieve all employees that work for a particular manager, including those farther down the chain of command. An example of this query, using the transitive closure construct is:

```
retrieve* into subord ( E.name, E.mgr )
      from E in employee, S in subord
      where    E.mgr = "Smith" OR
               E.mgr = S.name
```

This query first moves all rows of the employee table that have a manager named Smith. Then, the query looks at each name of the subordinates table and compares those to manager column of the employee table. All of those people work for a subordinate of Smith. This process continues until there are no more rows retrieved.

Transitive closure also works on append, delete, and replace commands. In each case, the query continues to operate until no more rows are touched by the command.

Rules

Rules allow the database to become active and respond to changes in the data instead of waiting for a user to submit a command. A rule tells Postgres to perform an action whenever a certain event happens in the database. For example, a rule can be constructed that ensures that a purchase order is never issued unless the vendor is already on the approved vendor list. The rule would have the following syntax:

```
never append to PURCHASE_ORDER where
      PURCHASE_ORDER.vendor != APPROVED.vendor
```

This rule would look at every append to the purchase order table. Before that append is allowed to proceed, the approved vendor would be consulted to see if there was a match of vendor names.

This type of a rule is a "never" rule. There are two other types of rules. An "always" rule is used to always perform a certain action when a condition occurs. A "once" rule is activated once when the condition occurs.

The purchase order example showed how a rule can be used to enforce referential integrity, which defines the integrity of a database object, such as a table or column, in terms of its relationship to other database objects. In most commercial database systems, referential integrity is the responsibility of the application, because most systems have no mechanism to tell the data manager how a certain table or column relates to another table or column in the database.

With QBF, for example, it would be quite easy to add an unapproved vendor to the purchase order table. When this situation occurs, most users develop a custom application using a tool like ABF that ensures that the referential integrity rules are enforced. The advantage of Postgres rules is that referential integrity is enforced for all applications.

Another example of referential integrity is the updating of derived data. A sales table might contain line items for each sale. A summary table might contain sales by salesperson. An always rule would be used to always replace the total sales by salesperson whenever a new line item was added to the base table.

Rules can also be used for security purposes. To restrict a user's access to data, the following rule can be submitted:

```
never retrieve ( EMP.salary )
      where username() = "Smith"
```

This rule would prohibit the user Smith from ever accessing salary information.

Rules are also useful as a way of alerting user programs when data change. For example, it might be necessary to notify the sales manager whenever a sale is recorded over a certain amount. A program could be written that stays resident on the system and submits the following rule:

```
always retrieve (SALES.salesperson, SALES.amount )
      where SALES.amount > 100000
```

Whenever a sale of greater than $100,000 is recorded, the program that submitted the rule would be notified. The program could then format and send an electronic mail message to the sales manager.

A one-time rule is used to perform an action only once. An example would be a customer service application. A customer calls up and says that she will be posting her bill late this month. The credit department could post a once rule so that it is notified when the payment is received.

One potential problem with rules is that they may conflict. For example, consider the rule that always sets Bill's salary equal to Mike's:

```
always replace EMP ( salary = E.salary)
        from E in EMP
        where    E.name = "Mike" and
                 EMP.name = "Bill"
```

The problem arises when another rule is constructed that always makes Bill's salary equal to Fred's. The two rules conflict. It is impossible to satisfy both rules. Postgres solves this problem by allowing the user to assign priorities to rules. If two rules conflict, the one with the highest priority is executed. If two rules conflict with the same priority, one is executed and the other ignored.

Transaction Management

One important difference between Postgres and existing database systems is that in Postgres data are not deleted (although a user can periodically purge a relation, which does actually delete the data). Instead, a time stamp is added that signifies at what point in time the tuple ceased to be current. This storage architecture has several important implications.

First, recovery becomes much simpler. Because old data are usually never deleted, there is never a need to go to a recovery log and undo or redo an operation. If the new tuple was successfully written to disk, it is valid; otherwise the old tuple remains current. For Postgres to function effectively in this environment, it needs to move data pages to stable storage for a change to be effective. If stable main memory is available, the pages are simply moved to that part of main memory. However, if sufficient stable main memory is not available, the pages have to be flushed to disk. Flushing pages to disk limits the effective speed of Postgres because the speed of a disk drive will act as an upper cap on the processing rate of the database system.

Because data are not deleted, it also becomes possible to perform queries on historical data. By default, a query operates on the current version of the data. Instead, a query can be issued to show what the data looked like at a particular point in time. For example, we can retrieve the names of all employees that worked for the company on a particular date:

```
retrieve (E.all ) from E in EMP["January 7 1985"]
```

It is also possible to retrieve data that were valid between a range of dates. For example, this query retrieves all employees that worked for the company in the last year:

```
retrieve (E.all ) from E in EMP["today" - "1 year" , "today" ]
```

Keeping historical data is useful for a wide variety of applications. For example, a sales forecasting staff may wish to evaluate the accuracy of a model. Forecasting models should be able to make better predictions the closer they get to an event. The model could be run based on data at various points in time to see if the accuracy of the forecasts increases as the data available become more accurate.

Another potential application would be a design engineer trying to decide why a component ceased to function effectively at a certain point in time. If the component is a complex object in Postgres, it can be retrieved at different points in time to compare the changes that were introduced in the design.

The basic advantage of keeping historical data is that in many situations users do not know what information they will need. When data are updated or deleted, information is lost; by keeping information, the database becomes more flexible and is able to answer a larger number of potential queries.

Versions

In addition to keeping historical data, Postgres allows the user to construct several versions of a relation. The information in the base relation does not change. The versions inherit all the data in the base relation and change existing data or append new data. When data in the version conflicts with the data in the original relation, the version sees the new information and the base relation sees the original data. The changed information in the version never appears in the base relation.

Versions are useful in situations where multiple people each need to see some private version of the database. For example, a code management system could be constructed using Postgres. Source code would be placed in a relation in the database. A new version would be constructed for each programmer, allowing each to change the source code of the system. When the code is debugged, the changes are merged back into the original relation.

Another possible application would be to save spreadsheets used for making forecasts in Postgres. Picasso could be used to retrieve the spreadsheet and display it on a form, as the spreadsheet is a type of table field. Other users could retrieve the spreadsheet and try different assumptions for the forecast. Each user would then have a different version of the same spreadsheet, and could even compare two versions to see how they differ.

Versions are created either from the relation or a snapshot of the relation. By default, the version created from a snapshot does not change. This means that any changes to the original relation do not show up in the version. If the version is created from the relation and not the snapshot, updates to the original relation also show up in the version (but not vice versa). A merge command allows two versions of a relation to be combined into a single relation.

Parallel Processors and Storage

Postgres has been designed to work on computers with multiple processors and large amounts of storage. It does so by using a three-level storage system, through the use of a novel method of storing data on secondary disk, and by allowing different portions of Postgres to run on different CPUs on a parallel processor.

Three-Level Storage System

Because data usually never get deleted and because a Postgres object can be potentially very large, the management of storage space becomes an important issue. Postgres uses a three-level storage system:

- primary storage (main memory)
- secondary storage (magnetic disk)
- tertiary storage (optical disk or magnetic tape)

Main memory is used to cache data in Postgres that are frequently needed, such as system catalogs and secondary indices. To prevent an I/O request for each of these objects, a cache manager is used to move data from the magnetic disk up into the cache area. The responsibility of the cache manager is to manage the limited space in main memory. Note that limited is a relative term. Because objects are potentially very large in Postgres, the designers assume fairly large amounts of main memory.

The magnetic disk contains the current version of the database, Postgres system and help files, and secondary indices. Each relation and each index is contained in a separate file. Since magnetic disk is a limited resource, tertiary storage is used to keep historical data.

A vacuum demon is used to move historical data from magnetic disk to tertiary storage. The vacuum demon looks for data tuples that have been superceded by new tuples. The vacuum demon also discards data from aborted transactions and other data that are no longer valid and updates the secondary indices to reflect the new location of the data. The purge command allows the user to specify how much historical data are kept. To only keep historical data after January 1, 1987, the following command would be submitted:

```
purge EMP before 'Jan 1 1987'
```

To purge data greater than one week old, another form of the command is used:

```
purge EMP before "@ 1 week ago"
```

The default, without a purge command, is to keep historical data forever.

It is also possible for a user to disallow historical access to data, in which case information is never moved to tertiary storage. When a relation is created, the user specifies the archive mode of the relation. An archive mode of "none" specifies that no historical access is allowed.

Separate indices are kept for secondary and tertiary storage of tables, allowing different types of indices to be constructed for tertiary and secondary storage. Since the media have different access and performance characteristics, different indices can help speed performance.

All of the index for a relation on magnetic disk and at least part of an index for a relation on tertiary storage is kept on the magnetic disk. This is because a magnetic disk is significantly quicker than tertiary media, such as optical disks. It is possible to have a portion of the index for the tertiary storage relation to also reside on tertiary storage.

Secondary Storage

The designers of Postgres are attempting to deal with two sets of design criteria in establishing a secondary storage system:

- efficient handling of large objects
- efficient recovery from disk failures

An assumption in the Postgres design is that large numbers of 3 ½" and 5 ¼" disks will replace the current large-capacity storage system. A typical system might thus consist of 100 or more fairly small drives. By allocating data across these disk drives, both performance and fault tolerance can be achieved.

Large objects can be striped across multiple disk drives. Striping means that different portions of the object are stored on different disk drives. Instead of going to a single drive, with its limited bandwidth, to retrieve the data, the system can take advantage of the bandwidth of multiple disk drives. Storing an object, such as a relation, on multiple disk drives also increases parallelism because multiple query fragments can each execute on different disk drives.

Putting data on multiple disk drives is also a way of preserving information in the case of a disk crash. On more traditional systems, fault tolerance is often achieved using disk shadowing, which entails writing every occurrence of one piece of data onto a second drive, thus doubling the amount of storage space used. An alternative approach used in Postgres is to make a single logical disk drive consist of multiple physical drives. A typical logical drive might have nine drives. The first eight have the actual data, and the last drive is considered to be a parity drive.

When a piece of data is read or written, it thus has to be looked at in two places—the actual data and the parity block. Careful buffer management helps eliminate much of the actual I/O involved. Also, to reduce saturating the parity drive, the parity bits are actually interleaved across all the drives. Careful design of this file system means that no one drive becomes the bottleneck.

The implication of this system is that the loss of any one drive is recoverable instantaneously. If a drive goes bad, the data can be recovered by going to the other drives with data and the parity drive and reconstructing the information. Although this takes longer than the original I/O with all drives functioning, it does not require a shutdown and restore of the system.

Reconstruction of the bad drive occurs in the background while normal data retrieval is progressing. The system interleaves reconstruction with conventional I/O. Although this is slower than reconstructing by shutting off conventional I/O, it does increase availability. Normally, the time it takes to recover a disk drive when all data are unavailable to users is fairly short. The interleaved strategy takes a longer period of time, but would have no unavailability of data during that time.

Support for Parallel Processing

The Postgres designers are attempting to support highly complex queries that involve a large amount of data. To do this effectively, the database system has to be designed to take advantage of a computing environment with parallel processors. In Postgres, the query optimizer is able to break a query into multiple parts to take advantage of available processors on a system.

Most of the Postgres functions have been designed as small asynchronous functions rather than one monolithic database server, which allows each function to migrate to different processors on the system as they become available. The query optimizer is able to break a query down into several pieces, each one being executed on a different processor. There are two types of parallel queries that Postgres recognizes:

- interquery parallelism
- intraquery parallelism

In a normal program stream, a single query is submitted to the back end and the results returned. For many queries this serial stream of control is unnecessary. Postgres allows several queries to be run in parallel, known as interquery parallelism. There are two mechanisms that allow interquery parallelism. First, the programmer can explicitly spawn several processes, each of which runs a query. The programmer then uses the normal interprocess communication methods on the system to collate the results. A more elegant solution allows a single program to submit multiple queries at the same time. The queries are separated with a single keyword, "parallel," which tells the run-time system that the queries can be run concurrently.

This solution for interquery parallelism requires the programmer to signal the back end about which queries can be run concurrently. A much more general solution, and also a significantly harder one to implement, would have the query optimizer analyze the semantic content of the queries to determine where parallelism exists.

Intraquery parallelism allows a single query to be broken into several pieces and run on different processors. A single query is thus broken into a series of plan fragments. Each of the fragments is submitted to available processors, and the results of the plan fragments are made available to the next plan fragment, until a final result is achieved for the query.

This is essentially what INGRES/STAR does in a distributed environment. INGRES/STAR breaks a query down into a series of locally sufficient queries and dispatches them to each of the local databases. INGRES/STAR then collates the results and performs final processing.

Summary

As can be seen from the fairly cursory description in this chapter, Postgres is a highly ambitious project to change the nature of database management systems, adding features that make databases active, high-performance and extensible. The goal is to allow complex objects to be managed using simple queries and thus manage a wider variety of data.

If data are all stored in a database, instead of scattered in files throughout the computer, there are several important implications. First, everybody requests data in a logical fashion. It is then the responsibility of the query optimizer to figure out how to get the data. Programmers do not need to spend hours developing their own storage structures, declaring variables, and handling routine input and output. Second, because data are in the database, services such as concurrency, security, and storage management are automatically provided.

To function effectively as a general-purpose data repository, Postgres needs to be as fast as any file system. Features such as precompilation, precomputation, parallel processing, and extensive caching of data in main memory are all features of the Postgres system to get fast access to data.

Postgres provides a glimpse of where commercial systems such as INGRES might move to in the future. It should be emphasized that Postgres is a research project and it will take some time (and some marketing decisions) before the features show up in any system, including INGRES.

Part III

Remote Data Access

Overview

Part III discusses how data can be accessed across a network. The data being accessed can be an INGRES database, in which case all of the components involved are INGRES components. For this reason, Chapter 8 is titled Homogeneous Data Systems. A homogeneous data system is able to use a highly complex, heterogeneous network. The networking protocols and computers are all masked from the components of the data systems, INGRES front and back ends.

Chapter 8 also discusses how several different databases can be combined to form a distributed database. The distributed database allows the user to remain unaware of the location of data. Information is presented as a series of tables. The location of the data on the network and the location within a particular database are transparent.

Chapter 9 extends this remote access to data by including heterogenous data systems. With INGRES/Gateways, an INGRES front end is able to access another vendor's back end. Gateways allow the INGRES toolset and applications to access a variety of different data repositories.

The reverse of the INGRES/Gateway is allowing another vendor's application to access an INGRES database. A wide variety of software vendors have the ability to store their data in an INGRES database. Chapter 9 discusses several of these products, such as spreadsheets, artificial intelligence systems, and natural language interfaces.

Chapter

8

Homogeneous Data Systems

The General Communication Facility

Up to this point we have made several different simplifying assumptions. The user interface and the data manager have been discussed, while the question of how the interface knows which server to connect to has been ignored, as well as the question of how the server and the application communicate with each other. Even with the simplifying assumption, we have seen that at least four different programs run to process a query. The application itself is one program. The data manager is another program, and is supplemented by the archiver and the recovery manager.

This chapter introduces the General Communication Facility (GCF), which allows servers and applications to find each other and communicate without regard to the underlying communications mechanism. At its simplest, GCF allows a server and application on one machine to communicate with each other, as well as allowing applications to communicate with servers on other machines without any change in the application code. GCF intercepts all commands from a server or application and hides the underlying details of the communication mechanism. The reason that applications and database servers are not aware of the underlying network is that they communicate with each other using a protocol called the GCF Applications Interface (GCA). All INGRES processes communicate with each other using the GCA protocol.

This chapter introduces two new processes that run in an INGRES environment. First, the name server keeps a list of the addresses of various database servers on the local machine and throughout the computer network. When an application wishes to access data, it first requests the address of a data manager from the name server, then initiates a connection to the appropriate server.

The second program is the communications server, which accepts queries and commands on the behalf of the application and then sends the data over a network to another communications server, which in turn delivers it to the database server on the remote machine. The communications server is able to function in a heterogeneous network. INGRES data servers can run on various types of computers. These computers, in turn, are connected to various types of computer networks. The communications server is able to work effectively over DECnet, TCP/IP, and the Systems Network Architecture (SNA), for example. In addition, the communications server can easily migrate into an Open Systems Interconnect (OSI) network as those protocols mature.

GCF allows any front end to communicate with any back end in a complicated network environment. The characteristics of the network are hidden from both the application and the data manager. Built on top of GCF is a further level of transparency, the INGRES/STAR distributed database environment. A distributed database allows several different databases to appear as one database to the user or application. This fundamental principal of a distributed database is known as Date's rule zero of distributed databases. The rule, developed by C. J. Date, is simple—the user should see no difference between a distributed database and a nondistributed database.

When a user submits a complex query, it is possible that data from several different locations will be joined together. The distributed query optimizer breaks the query down into several locally sufficient queries that are dispatched to the appropriate server and the results returned. Processing continues by taking the partial results and merging them into the final result requested by the user.

A distributed database, because it uses the communication services of GCF, can consist of databases that operate on different types of computers. The corporate mainframe can have one database on an IBM computer, a department might have another on a VAX, and an individual might have a third database on a MicroVAX or Sun Workstation. All three of these separate databases appear as a single logical database to the user of INGRES/STAR.

This chapter discusses various levels of distribution in an INGRES/STAR environment. It is a fairly simple process to have a distributed database do only retrievals on the target local systems. INGRES/STAR is able to provide a higher level of distribution, updating remote systems and processing queries in an efficient manner so that the entire remote table is not returned to the distributed server for processing.

This chapter discusses a homogeneous database environment. Homogeneous in this case means that all the programs involved are INGRES programs. Note that the underlying network might be heterogeneous: Each of the data servers might live on different operating systems and use a variety of different networking protocols. The next chapter deals with heterogeneous database environment, as in the case of an INGRES front end accessing data from a non-INGRES database system, such as DB2 or IMS. Once again, the underlying network architecture can also be heterogeneous.

The GCF Application Interface

All INGRES components use the GCF Application Interface (GCA) for communicating with other components. GCA allows data to be transferred in terms of native relational database constructs, which include:

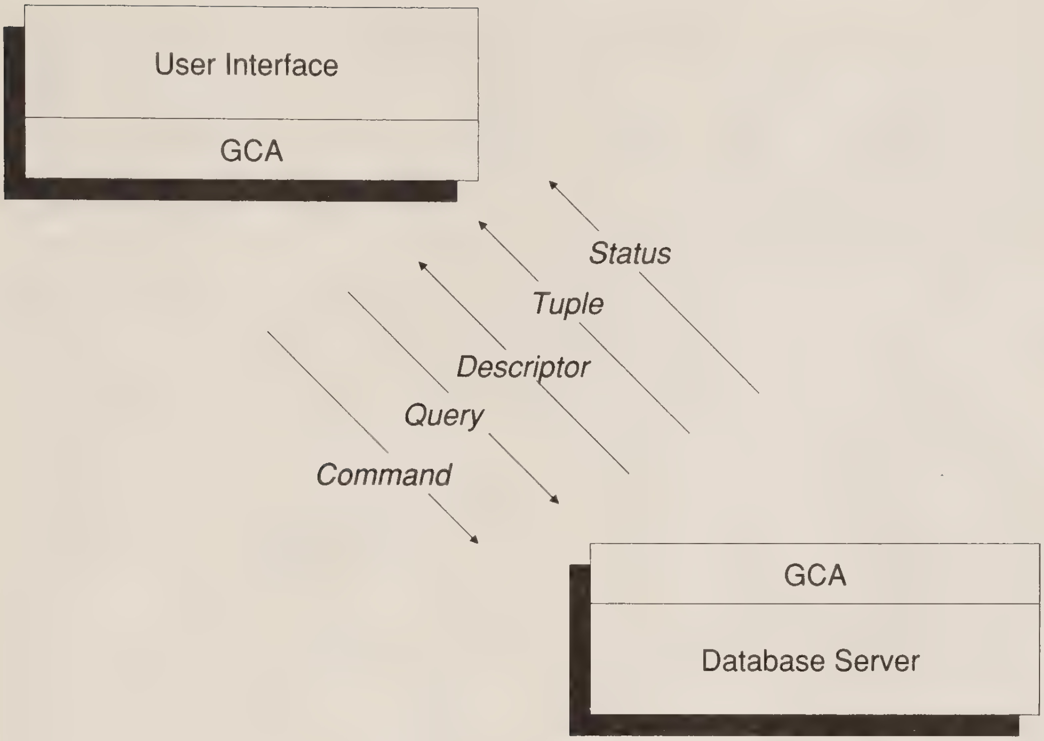


Fig. 8-1 GCF Application Interface

- commands
- queries
- tuple descriptors
- tuples
- status messages

An application would send a query to the GCA portion of the INGRES system. GCA would then dispatch that query either to a local data server or to the communications server. When the query finally reaches the destination data server, the GCA interface for that server presents the query to the query processing portion of the server. The server would then return either data or status information back to the GCA interface. Figure 8-1 illustrates this process.

Two types of data are returned when a successful query is processed. A tuple is simply a row of data. Each row may have several columns of data in it. A tuple descriptor, sent by the database server before the tuples, describes what those rows look like: the name, data type, and length of each of the columns. When a dynamic SQL program submits a query, it may use the tuple descriptors to allocate local storage before accepting the tuples themselves.

The GCA interface can take a very large message (such as many tuples) and break it into several pieces. The pieces are then reassembled at the remote side. Segmenting a message might be necessary if the application or communications server has a small amount of buffer space and cannot accept all the data at once. GCA also has a provision for expedited data. Let us say a user requests all 10 billion rows of data from a personnel table and dispatches the query. Then, the user realizes that it would take a few minutes to examine all these data and a more selective query might be more appropriate. The user types an interrupt key to abort the query. It does not make sense to deliver the interrupt command after the entire query is processed, so GCA sends the interrupt as expedited data so that it is quickly presented to the remote data server.

When a user starts up INGRES/MENU, he does so with a target database name. INGRES/MENU uses GCA to start up an association with a data server. Next, the user will pick some other facility such as QBF. QBF and INGRES/MENU are different programs on a computer and might thus be required to each establish their own session with the data server. However, GCA is able to have a child process (QBF) inherit an association with a data server from a parent process (INGRES/MENU) to avoid the proliferation of idle sessions.

GCA is also able to modify the characteristics of an association in midstream. With INGRES/MENU, a small amount of memory is required, because very little data are used. If the user then starts up an application that has many different forms and will return a lot of data, that application will allocate a larger amount of memory. It makes sense for this application to use part of the memory to increase the buffer size for data coming back from GCA.

The importance of GCA is that it isolates all INGRES processes from the details of communication. A program like QBF can be enhanced without worrying about the particular implementation of QBF on a network or computer. Likewise, the networking capabilities of INGRES can be enhanced without an effect on the applications or data servers.

The Name Server

When a program requests the use of a particular database, the first step is to contact the name service process with the name of the target database. The name server translates the logical database name into an address for a server. In the case of a local connection, this may be the process ID for the server.

All servers on a system initially register themselves with a GCF address. Periodically, the IIGCN (INGRES GCF Name Server) process goes to each of the data servers and asks if it is still offering that particular service. The II Nameserver Utility, activated by typing “IINAMU,” is used to show the current status of registered servers. The *SHOW SERVER* command gives a list of servers, the database they offer services for, and the GCF address of that device.

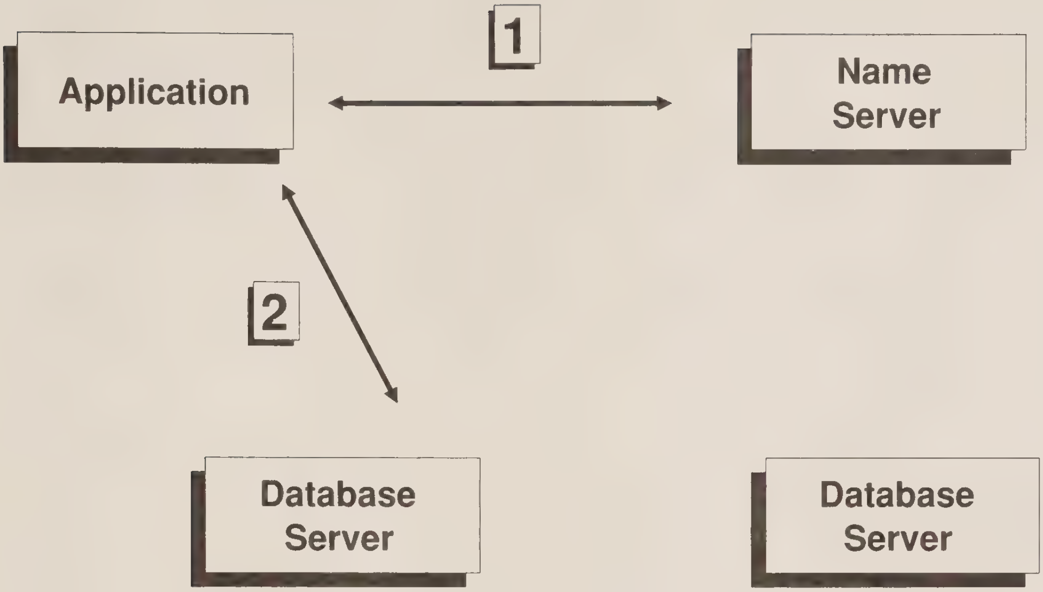


Fig. 8-2 Single Machine Components

The address for a local data server is simply the process identification number or address for that server. The application uses the interprocess communication (IPC) facilities of the local operating system to initiate communication with that server. Figure 8-2 shows the operation of INGRES components on a single machine. Notice that there can be several database servers available, but the application can only communicate with one server at a time. If the server is remote, the application uses the local IPC facilities to contact the communications server. The communications server then handles the network-based communications on behalf of the application.

The GCF Communications Server

On a single system installation of INGRES, there are five types of programs running:

- applications
- data servers
- a name server
- a recovery manager
- an archiver

If the implementation is running on a DEC VAX Cluster, there is also a cluster coordinator process running that keeps the various local transaction logs synchronized. The VAX Cluster looks like a single computer as far as the data server is concerned. This is because it is able to use the local file system to access data. All nodes of the cluster see a common file system. Once the data are retrieved, they can be easily moved to the application using the local IPC facilities of the computer.

Over a network, communication between an application and a server is more complex. Instead of the local IPC facilities, the programs need to use the services of the network. The network sets up a virtual connection to the remote system. Setting up a virtual connection differs in different networking environments. The communications server is the program in the INGRES environment that provides the interface to the underlying network.

For purposes of this section, we ignore how the underlying network is put together. There may be various data links such as Ethernets or high-speed leased lines and there may be various computers in the path between the two target systems, used by the network as intermediate nodes for routing data. These lower-layer network issues are discussed at the end of this section.

For this section, we assume only that the network is providing a transport service. A transport service allows two programs on the network to communicate with each other and is responsible for providing reliable end-to-end communication—all the data sent by one program are received by the other program, in the order in which they were sent.

GCF builds on this transport service to provide higher levels of functionality. The implementation of GCF, the communications server, has four layers, each building on the services of the underlying layer. These four layers are

- the application layer
- the presentation layer
- the session layer
- the transport layer

The four layers of GCF are the four upper layers of the International Standards Organization's OSI Reference Model. The OSI Reference Model also includes three lower layers, the network, data link, and physical layers. GCF, in combination with the transport service, provides a full implementation of the OSI Reference Model.

The bottom part of the communications server is the interface to the transport layer of the network. This lowest layer initiates a virtual connection to the remote node. If the underlying network is DECnet, for example, the server uses the End-to-End Communications process, which is DECnet's transport layer. If the underlying network is TCP/IP, the communications server initiates a session using the Transport Control Protocol, also a transport layer service.

The bottom layer of the communications server contains the network-dependent functions of GCF. Different modules in the transport layer of the server are used to initiate virtual connections on TCP/IP, SNA, and DECnet. Adding support for another network, such as the Open System Interconnect (OSI), consists of writing the interface module at the transport layer.

The session layer of the communications server uses the services of the transport layer. The transport layer offers a service: reliable end-to-end communications. The session layer manages the characteristics of a particular session: validating user access and handling session aborts.

Built on top of the services of the session layer is the presentation layer, which is responsible for converting data that are in a different format from the current system's. An integer, for example, has a different representation on a VAX and on an IBM computer. On one, the first bit of a byte is most significant; on the other, the last bit is most significant. If we send a 1 from one computer, the other computer interprets the information as 256. To be usable, the data must be converted. Character data also have different representations on different systems. On an IBM System/370 computer, the EBCDIC character set is used. On a VAX, the ASCII character set is used. The function of the presentation layer protocol is to translate data from one format into one that can be understood by other systems.

The top layer of the protocol stack is the application layer. GCA is the application layer protocol. Data are sent to GCA in terms of messages, which format the tuples into a buffer. The presentation layer translates the message into the proper format, the session layer initiates a session, and the transport layer sets up a virtual connection for the session.

The advantage of a layered protocol is that new functionality can be added to one of the layers of the protocol stack without rewriting the other layers. For example, the session layer might be enhanced to provide recovery services. The recovery service would make periodic checkpoints of the session so that in the case of network or computer failure the session could resume at a later point in time. The presentation layer thus would not have to be rewritten because it does not provide session services; it simply translates data.

The network implementation of GCF, marketed under the name INGRES/NET, is the communications server. When an application wishes to use a remote database, it first sends a message to the name server. The name server returns an address, consisting of the address of the communications server, the network address of the remote database server, and information on the transport service that is to be used. The application uses the GCF Application Interface to set up a session with the communications server. The communications server then sets up a virtual circuit over the network to the communications server on the destination node. Note that there could be intermediate communications servers in the path to the final destination in a heterogeneous network architecture. Once at the final destination, the communications server uses GCA and the interprocess communication facility to send the connection request to the database server. Figure 8-3 illustrates this process.

The system management interface to the communications server is a program called NETU (for INGRES/NET Utility). This program allows the system manager to control the functioning of INGRES/NET. The INGRES manager is able to use NETU to set up network connection information. For each remote server, the manager enters the information needed to establish a session. Individual users can use NETU to tell the communications server their user name and password for foreign systems. INGRES/NET thus

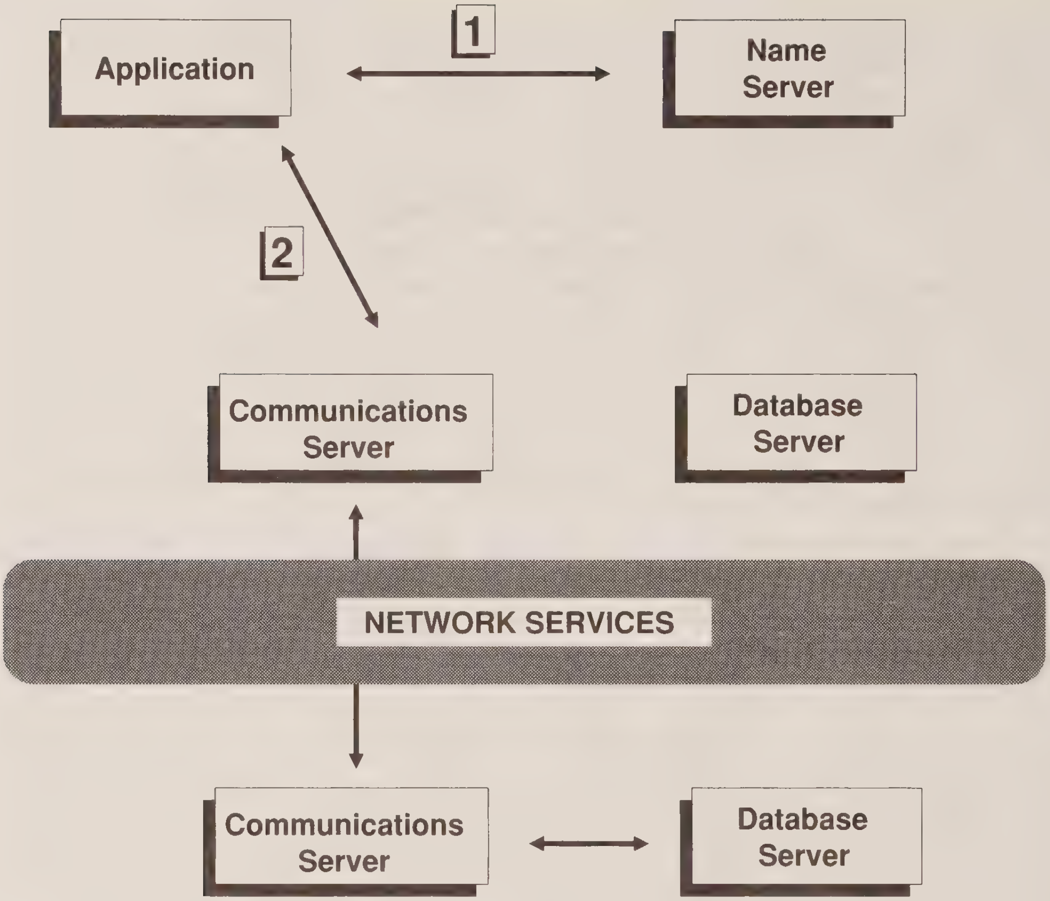


Fig. 8-3 INGRES/NET

preserves security on different systems by only allowing authorized users to access the remote systems.

A network conceptually consists of a series of computers with communications lines connecting them. Either of these components can be of varying power and have varying degrees of utilization. The system manager is able to use NETU to enter weighting factors for both computers and communications lines. The weighting factors are used by the INGRES/STAR distributed query optimizer to decide how to most efficiently process a query. If a remote computer is very slow, or the communications line is slow, the query optimizer will try to process as much of the query as possible using more efficient systems.

NETU allows the user to set up private configurations of remote nodes and communication lines. A private configuration is only used by a single user or a limited group of users. For example, a group of users might have a departmental MicroVAX with a database on it. The MicroVAX can be defined on other computers on the network as a private configuration.

With INGRES/NET, the same application can be run against different target databases. Instead of typing *QBF database_name*, the user simply types *QBF node::database_name*. The application contacts the name server, which consults its list of known servers and then identifies a network address for the remote data server.

INGRES/NET provides an important level of flexibility in designing applications. Most organizations will have several different databases in different locations. With INGRES/NET, the same user interface can be used to access any of these remote environments.

Role of the Underlying Network Architecture

GCA shields the application and data manager from the intricacies of communication across the network or on a computer. In turn, the transport layer of a network shields GCF from the intricacies of the underlying network environment. Remember that the transport layer provides reliable end-to-end communications for GCF. Messages are always received in the order sent and the transport layer guarantees that all messages sent will be received.

Under the transport layer are three more layers of the network:

- the network layer
- the data link layer
- the physical layer

Although the lower three layers of the network are shielded from the database system by GCF, it is important to understand some aspects of the underlying network for performance reasons. Although GCF can work over any network topology, as long as the appropriate transport mechanism is provided, it is important to match the configuration of the network with the anticipated data requirements.

The physical layer is responsible for taking a queue of bits on one computer and sending them to the other computer on the other side of the physical link. The physical layer only deals with controlling wires and modems and sending a queue of bits over the physical link. It does not concern itself with issues like the eventual destination of the data, who the data are for, or what the data will be used for.

The data link layer takes the service offered by the physical layer—sending bits over a link—and groups the data into frames, which are packets of information. The data link layer does not worry about the user of the data or their eventual destination. It simply offers a service of taking a frame it receives and sending it over a physical link. Examples of data links are Ethernet and Token Ring. The data link layer for Ethernet or token ring takes a frame of data and delivers it to the appropriate destination computer connected to the same media.

The network layer takes each of the incoming frames and decides if they need to continue through the network; it either routes data through the network or passes them up to the transport layer. The transport layer decides which user the data are for. In our case, the user would be the GCF communications server. The communications server then takes the data and sends them on up the protocol stack.

If the data are not local, the network layer continues to route them through the network. Many network layer protocols, such as the DECnet routing layer, are able to adapt to changes in the topology of the network. If a particular connection goes down, the network layer looks for another path to the destination node.

The implication of these lower three layers is that the transport layer need not concern itself with the topology of the network or the data links that make up the network. The transport layer looks for sequence numbers on messages to make sure all pieces of data have been received. If not, it requests that the transport layer process that it is communicating with resend that particular piece of the message.

Because GCF is built cleanly on the transport layer interface, the underlying network can be easily changed without changing the configuration of the data access mechanism. Changes in topology and data links add performance and redundancy but do not affect GCF, which is built in such a way that it is highly adaptable to different communications architectures and protocols. If the network manager installs a new high-speed data link, GCF does not need to be modified because it uses the carefully defined services of the transport layer.

GCF is also able to work quickly with new network architectures. To work on SNA, for example, the only part of GCF that had to be provided was the interface to the SNA transport layer. As networks based on the OSI protocols mature, GCF can easily be adapted to use the TP4 transport layer protocol of OSI.

This simple interface to the underlying network architectures is important because networks continually change. For example, several new data link technologies are emerging, such as FDDI, which operates at a speed of 100 million bits per second (mbps). Ethernet, by contrast, operates at 10 mbps. As network architectures such as DECnet move to incorporate FDDI, GCF will be able to operate over these higher-speed data links.

Another important development in the lower layers of the network is the Integrated Services Digital Network (ISDN). ISDN allows bandwidth to be easily allocated in a wide area environment. Currently, high-speed bandwidth requires leasing a dedicated line. This arrangement has a long lead time and is not flexible to changes in bandwidth requirements. As networking architectures such as DECnet become compatible with ISDN, GCF will be able to use these services. When a lot of data need to be transferred, additional bandwidth can be requested of ISDN and then released after the data transfer.

OSI also includes precise definitions of the upper layer of the network. The protocols used in GCF can be replaced with the standard OSI protocols. Thus, as the OSI presentation layer matures, Relational Technology can substitute the Abstract Syntax Notation of OSI for the current presentation layer of GCF. The application layer of OSI includes a Remote Data Access (RDA) component. This international standard allows applications in an OSI environment to work with SQL-compatible data servers throughout the network. These databases can be from any vendor, as long as they conform to the RDA standards. The application interface to GCF (GCA) is compatible with a subset of the RDA standards. As the standards mature, the INGRES architecture can still operate in this type of heterogeneous environment.

Distributed Databases: INGRES/STAR

INGRES/NET allows any front end to communicate to any back end over a heterogeneous network architecture. INGRES/STAR builds on top of the services of INGRES/NET to provide the next level of transparency. Using INGRES/NET, a user can communicate with a server, but must know where that server is located. The application can only communicate with a single server. The INGRES/STAR environment makes multiple local or remote databases appear as a single local database.

In a single application to server environment, we saw two tiers: the application and the server. INGRES/STAR adds a midlevel tier in between a database and the application. This midlevel is called the distributed database server. As shown in Figure 8-4, the user application connects to the distributed database server, just as it would connect to a local database server. The distributed database server has links to local database servers, which in turn interact with the databases.

INGRES/STAR has the characteristic of making many databases appear as a single database. It is thus possible to make this distributed database part of yet another distributed database. INGRES/STAR allows nesting of distributed databases up to 16 levels deep. The advantage of the Star architecture is that a particular local database continues to function as before. Applications can be written that interact with that local database. The local database can also participate in one or several different distributed databases. The architecture thus preserves the local autonomy of a database and allows for local administration of different data repositories.

An application treats a distributed database just like a local one. First, the application sends the name of the database to the name server. The name server returns the address of a data server. The application then sends queries and commands to the server using messages in GCA format. Figure 8-5 shows the process structure for an INGRES/STAR environment. Notice that in the diagram the distributed database server is on the local node. With INGRES/NET, there is no reason why the server could not be located on a remote node.

The distributed database server also uses the services of INGRES/NET to contact local database servers on remote nodes. If the local database server is on the same node as the distributed server, interprocess communication is used instead of INGRES/NET.

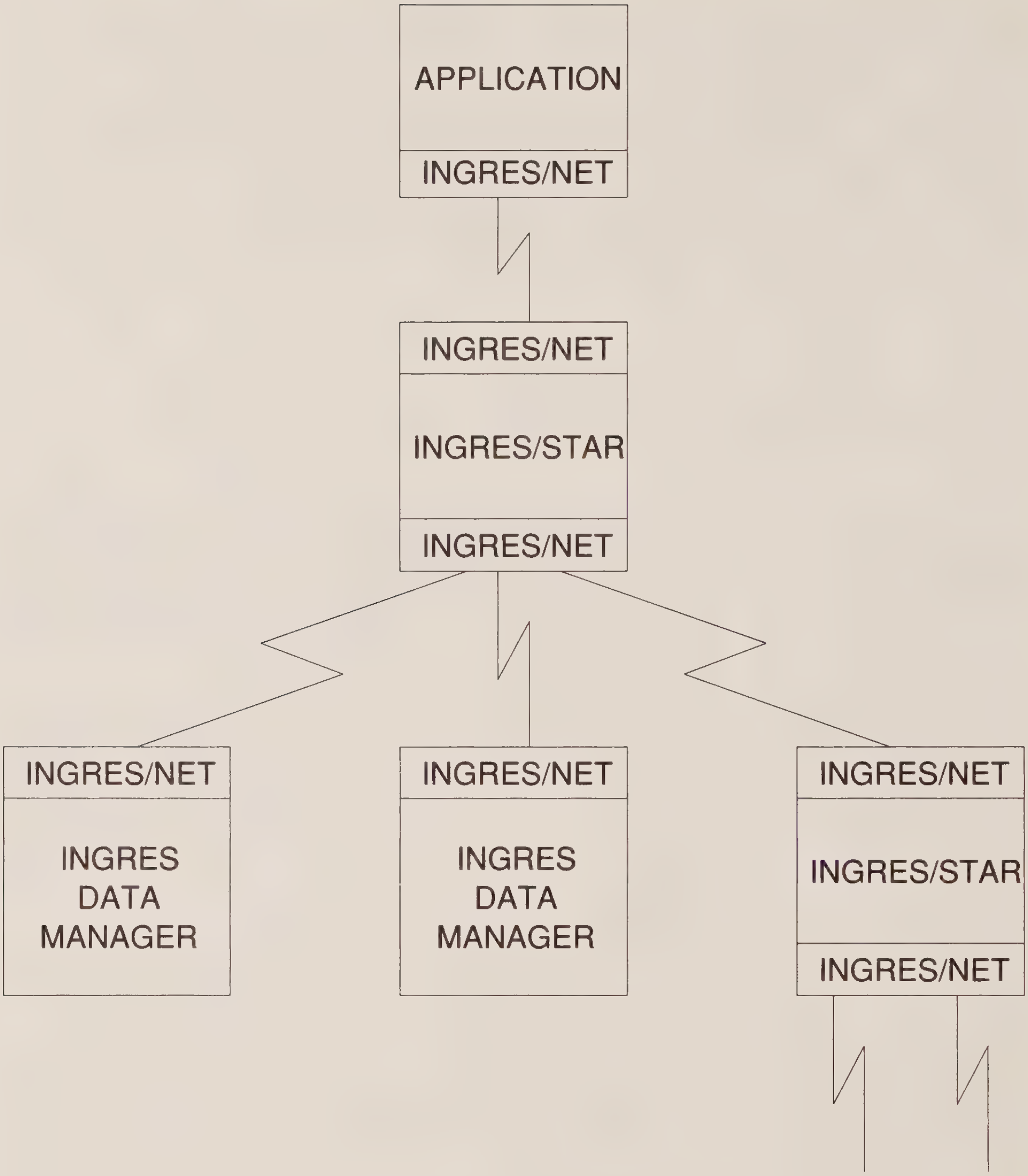


Fig. 8-4 Three-Level INGRES/STAR Architecture

To manage the components of a distributed database, the INGRES/STAR process uses a local database as a coordinator database. This local database contains tables and views and all of the other components of an INGRES database. In addition, the coordinator database contains information about objects contained in other local databases that form the distributed database. Tables and views in the local databases are registered in the coordinator database as a link (see Fig. 8-6). Note that to the user of the database

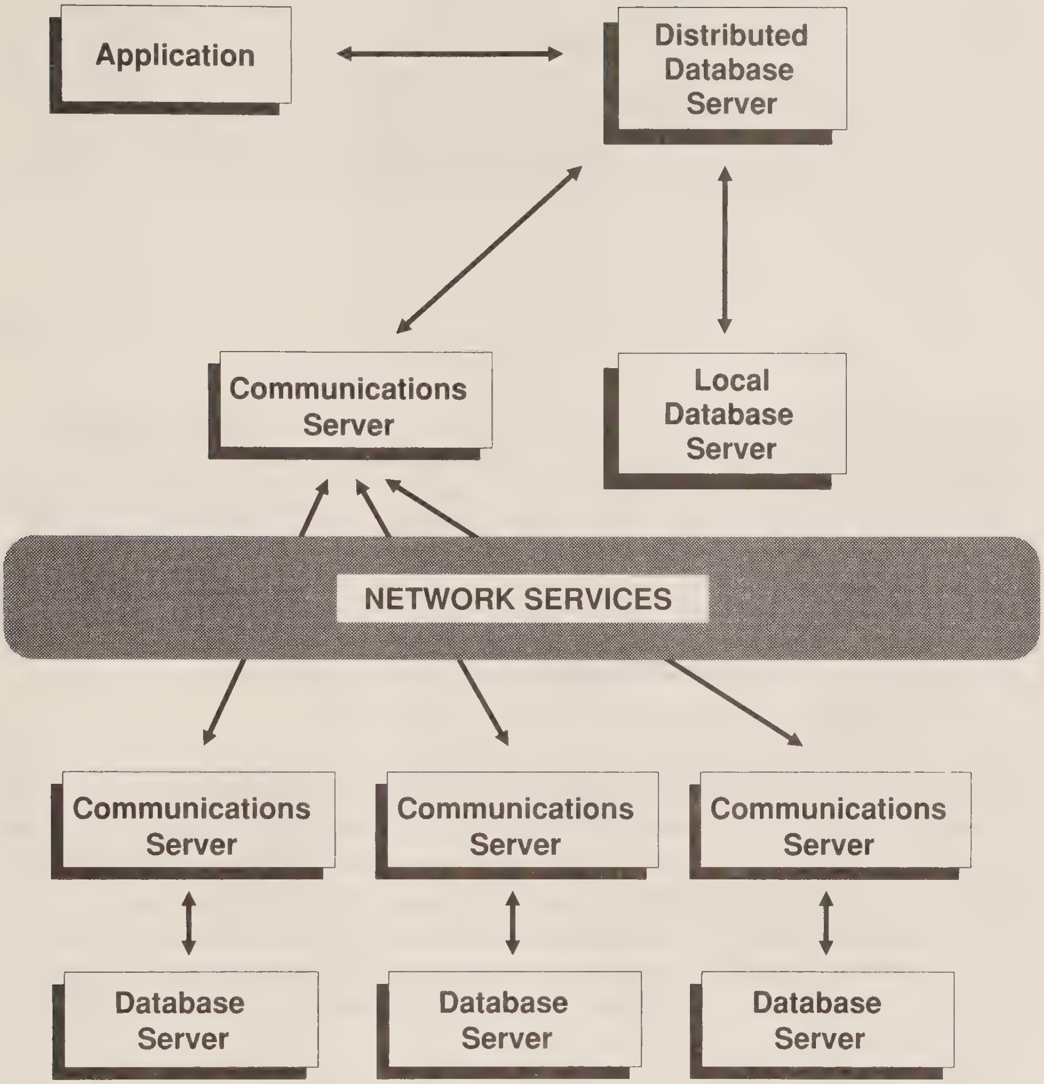


Fig. 8-5 INGRES/STAR

the links are transparent—they appear as views, tables, indices, and other components of an INGRES database.

There are thus three types of databases involved in an INGRES/STAR environment. The distributed database is the collection of all databases that appear to the user as a single data repository. There are a series of local databases that take part in the distributed database and may also take part in other distributed databases. Finally, there is the coordinator database, which is a special instance of a local database containing information about the configuration of the distributed database.

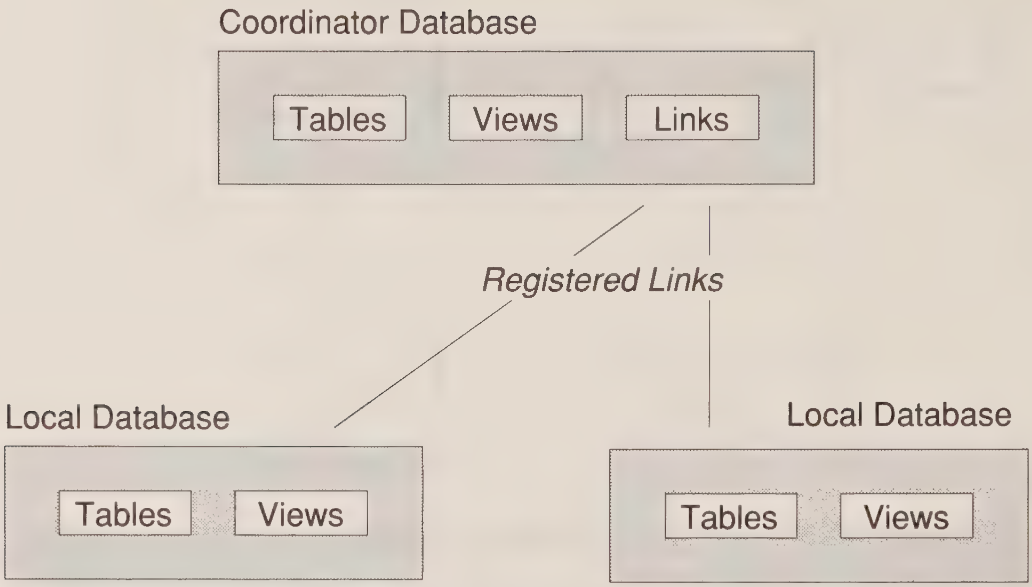


Fig. 8-6 INGRES/STAR Coordinator Database

An object in a local database is not necessarily visible to the distributed database. This allows information not pertinent to the wider, distributed environment to be kept strictly local. To declare an object in a local database as part of the distributed database, it must be registered. The *register* command takes an existing object and adds information about the link to the distributed database catalogs. It is also possible to create a table in any of the local databases that form the distributed environment. To create a table as part of the coordinator database, the user might issue the following command:

```
create table employee (    emp_name = varchar,
                           salary = integer )
```

To have the same table stored in another local database, the user could issue a variant of the create command:

```
create table employee (    emp_name = varchar,
                           salary = integer )
                 with
                           node = 'foreign_vax',
                           database = 'sales_database'
```

This command has two effects. First, a table is created on the node FOREIGN_VAX in a database called SALES_DATABASE. Second, the table is registered in the coordinator database.

A utility called STAR*VIEW is available for the distributed database administrator. This utility shows all of the databases and links that make up the distributed database environment.

Performance

Query optimization becomes significantly more complicated in a distributed environment. A query can involve the services of several local data managers. The optimizer has to decide what order to send requests out and how to combine the results into the finished query. The optimizer also has to take into account communication speeds and the amount of data going over communication lines.

High performance in a distributed environment is achieved through several mechanisms that allow the query optimizer to make informed decisions. When a query is received, the optimizer first looks for the location of the data. Then, it has to decide the best way to get the data. The query optimizer in either a local or distributed environment has a variety of pieces of information on how tables are physically stored. This includes the storage structure of the table, the key columns, and secondary indices; it may also include histograms or other statistics describing the profile of the data (see Fig. 8-7).

When the INGRES/STAR process connects to a distributed database, the coordinator database may not have the most up-to-date information on tables in local databases. A series of queries are dispatched to the system catalogs of the target local databases and this information is used to update the system catalogs of the coordinator.

In addition to updating the system catalogs, this information is cached, just like any local table descriptions. Because the cached information is stored in main memory, it is available to the query optimizer at much greater speeds than by going to the disk drive (or a remote system) to retrieve it.

The query optimizer has a variety of different query execution strategies available for most queries. It looks at the selectivity of a query and the relevant access methods to decide in what order to process a query. INGRES/NET is able to provide information on the processing power of various nodes and the relative speed of the different available communications links. Using this information, the query optimizer looks not only at how much data a given plan will involve, but how long it would take to move them from one location to another.

The optimizer is able to optimize a query for either response time or resource utilization. These two goals often conflict. Heavy use of network bandwidth can generate a quick response time, but often leads to high resource utilization as the computers involved have to process data at a faster rate.

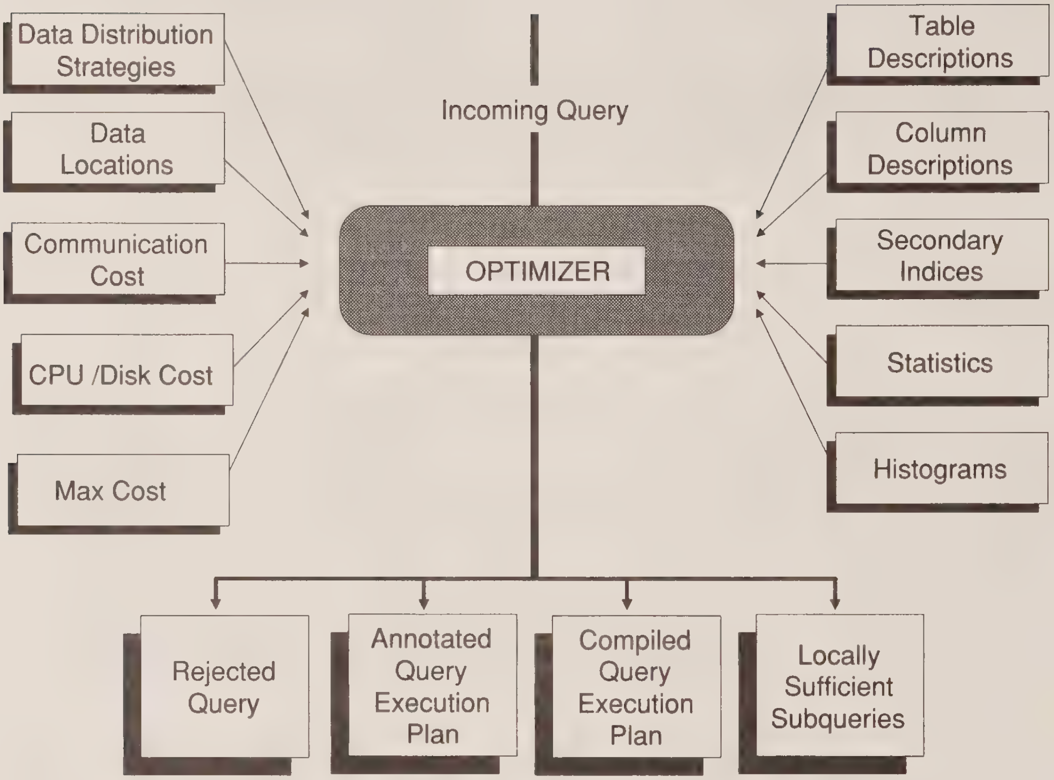


Fig. 8-7 Distributed Query Optimizer

The final output of the query optimizer is a QEP. Unlike the local environment, the distributed query optimizer formulates two sets of plans. The master plan is used by the coordinator node. A set of locally sufficient subset queries, or plan fragments, are dispatched to the relevant local servers. Each of these local query plans is processed by a local database server. The local server decides what locks to take and coordinates this plan fragment with other users of the server. Note that other users might be local users or another distributed database access. When the data are retrieved, they are sent back using GCF to the coordinator node.

One interesting feature of the INGRES/STAR distributed query optimizer is that users are able to set the maximum cost for a query. If the best available QEP exceeds the maximum cost, the query is rejected. This feature prevents an inefficient query from being dispatched over the network.

As with the local database server, the user is able to examine the QEP. For production applications, it is important to examine how a complex query is being executed. If a large amount of data has to be moved from one location to another, it might make sense to relocate one of the tables.

Levels of Transparency

Distributed databases can operate at various levels of transparency. Transparency is the issue of how well the collection of databases that make up the distributed database are able to emulate the functions that are available in a single local database. A low level of transparency is a retrieve-only distributed database. Higher levels of transparency allow distributed updates to be performed from within a multistatement transaction.

The question of multisite updates is one of the more important transparency issues. An example of a multistatement transaction would be moving money from a checking account table to a savings account table. There are potentially three databases involved in this transaction:

- the distributed database server
- the checking account database
- the savings account database

The purpose of the multistatement transaction is to make sure that all parts of the transaction are accomplished or that none are. The distributed database coordinator thus needs to make sure that each of the local databases is able to perform the action and in fact did perform the action.

The protocol used for distributed multistatement transaction is called a two-phase commit protocol (see Fig. 8-8), which ensures that distributed transactions remain synchronized and consistent. It is invoked when multiple sites are updated during a single transaction. First, the distributed database coordinator sends a message to the local databases asking them if they will be able to perform the relevant operation. Each of the local databases then secures the resources needed to commit the transaction by logging status information. The local database then returns a confirmation message to the coordinator.

When all confirmations have been received, the first phase is completed. At this point, the transaction is ready to be committed. The coordinator then tells each of the local databases to perform the action. When the action is performed, each local database sends back another message saying that the action has in fact been completed. Only when this second phase has been successfully acknowledged at all nodes is the transaction committed.

If one of the nodes crashes, the coordinator sends a rollback message to the other nodes. This ensures that either the whole transaction is committed or none of the changes are made. Just as a local server has a transactions log, so does the distributed database. The recovery manager for the distributed database is able to detect aborted transactions and roll back the relevant transactions.

Distributed databases can offer further levels of transparency. Two important research areas are:

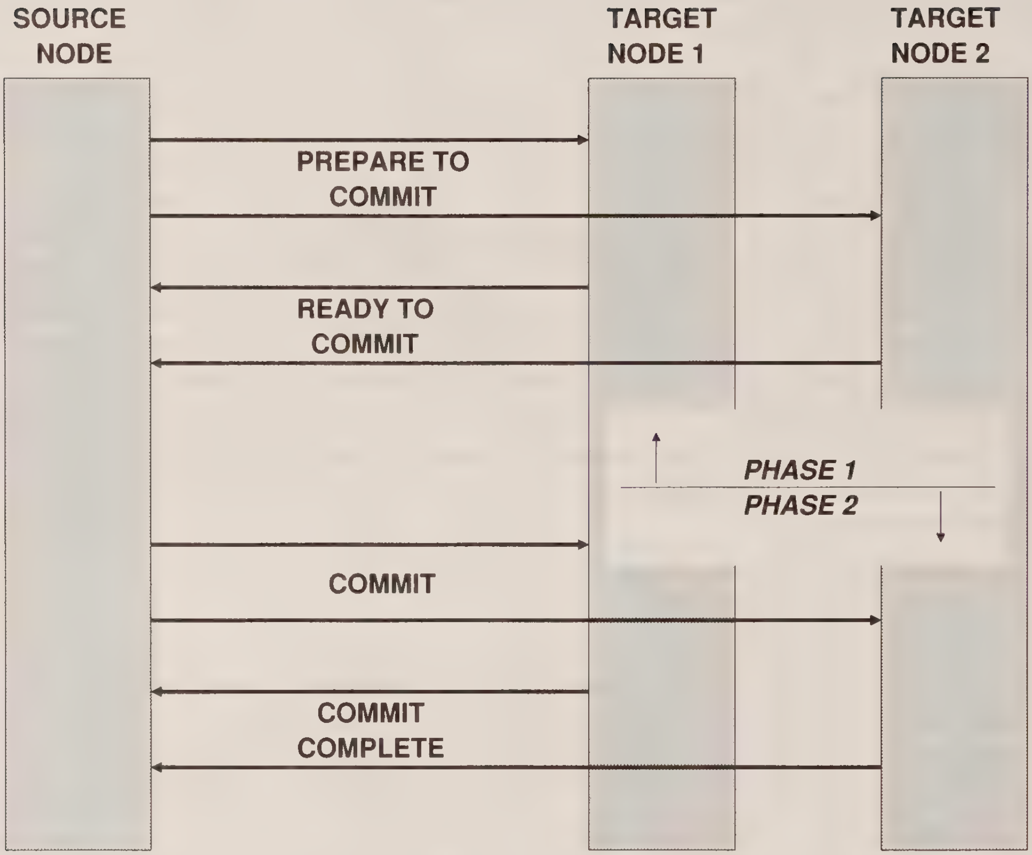


Fig. 8-8 Two-Phase Commit Protocol

- horizontal and vertical fragmentation
- replication transparency

Horizontal and vertical fragmentation allow a single table to actually reside on multiple databases. Horizontal fragmentation allows different columns of a single table to be partitioned into different databases. For example, a personnel table might have columns with salary and office location. The salary column could be stored on the personnel database, while the office location column could be stored on the building maintenance database.

Fragmenting a table allows the database administrator to keep data closest to those users who have the most frequent need for the data. The building maintenance staff would access the office location data more frequently than the staff in the personnel department. Note that users are unaware of the fragmentation. In all cases, they simply ask for data using SQL statements. The only effect of fragmentation is on performance.

Vertical fragmentation allows different rows of a table to be on different nodes. For example, a sales staff might have separate nodes for each of the different sales regions in a corporation. The eastern region sales staff could keep their data available on their local node. The same table could be vertically fragmented, with the western region rows on a different node of the network. Senior management can refer to the sales table as a single table, and the distributed query optimizer will retrieve the data from the different local databases and combine them.

An even further level of transparency is known as replication transparency. In this type of environment, the same data actually reside in multiple locations. If one node goes down, then the data are still available on another node. The query optimizer is able to decide which of the particular nodes is most appropriate for processing a particular query by deciding, for example, which node can provide the fastest response time.

With replication transparency, update and recovery of data becomes even more difficult. If a node is taken off the network or crashes, when it restarts it must first go to the other copies of the data to refresh itself. Whenever data are changed, the change must be made on all nodes. If the changes are not made in all places, it is possible that different users will see different values for the same data.

Summary

When a front-end process requests a connection to a back-end database, there is really no way for the front end to know which server is appropriate. In addition, it is possible that the front end and the back end reside on different nodes of the network. GCF is used by all components of INGRES and hides the details of communication from both front and back ends. These details include the problems of buffering data on a network, transformations of data representations across machine architectures, the various transport mechanisms used to segment and reassemble data, and a host of other issues.

The GCF architecture consists of three parts. First, the GCF Application Interface (GCA) is used to format messages for all INGRES processes. GCA is built into all INGRES tools, all servers, and is a part of the ESQL and EQUEL programming libraries. Second, the GCF Name Service (GCN) provides a name server to find the location of local and remote servers. Any INGRES component wishing to initiate a connection with a server sends the name of a database to the name server. The name server returns an address for the appropriate server.

Finally, the GCF Communication Service (GCC) is the network communication component of INGRES/NET. An application wishing to communicate with a back end on another node sets up a connection with the communications server. The communications server formats messages for transmission over the network and delivers the message to a communications server on the destination node. That communications server sets up a connection with the requested database server.

INGRES/STAR builds on top of the GCF services to add a further level of transparency. A distributed database can be created that has links to a variety of local databases. INGRES/STAR allows multiple data repositories to appear as a logical whole to the user.

Because of the approach of using a coordinator database, several important benefits result. First, the coordinator database supports a distributed query optimizer that is able to efficiently access the distributed data. Second, local autonomy is maintained because the local data manager still manages the local database. Third, a single local database can participate in many different distributed databases.

The next chapter will examine an extension to this environment. With INGRES/Gateways, non-INGRES data managers can offer services to INGRES applications. These gateways can be directly accessed through INGRES/NET and can also participate in an INGRES/STAR distributed database.

Chapter

9

Heterogeneous Data Systems

Kinds of Gateways

The previous chapter discussed how a series of INGRES data managers could make their services available over a network. This network could be heterogeneous, using several different networking protocols such as DECnet, TCP/IP, and SNA. The users of the General Communication Facility (GCF), however, were all INGRES programs. Thus, the previous chapter discussed how homogeneous data systems were able to use a heterogeneous networking environment.

This chapter discusses how heterogeneous data systems can function in the same environment. There are two very different types of heterogeneous data systems. The INGRES/Gateway products allow INGRES tools and user applications to access non-INGRES data sources. There are two types of INGRES gateways. The SQL gateways allow access to a relational database, such as IBM's DB2 or DEC's Rdb, which are accessed either directly as a back end from a front-end tool, or as part of an INGRES/STAR distributed database.

The other type of gateway implementation is for non-SQL systems. A non-SQL gateway allows traditional file systems and nonrelational databases to be transparently incorporated into the INGRES environment. These gateways can access file systems such as DEC's Record Management Services (RMS), or IBM's Virtual Sequential Access Method (VSAM). Nonrelational databases include IBM's IMS product, based on the hierarchical data management model and Cullinet's IDMS/R, based on the CODASYL model of data management.

The other type of heterogeneous data connection allows another vendor's application to access an INGRES database. Spreadsheets, knowledge bases, statistical analysis software, and natural language interfaces are a few of the products that are able to use the services of INGRES to store and manage their data.

INGRES/Gateways

A heterogeneous data system means that one vendor's application is able to communicate with another vendor's data manager. The INGRES/Gateway is meant to allow an INGRES user to access non-INGRES data in a transparent fashion. This user could be

- a general-purpose front end, such as QBF or the Report Writer
- an application developed using the ABF environment
- an Embedded 4GL program
- an INGRES/STAR distributed database server

If the user of the SQL gateway is INGRES/STAR, there is one more level of user—the application or program. With INGRES/STAR and gateways, INGRES data and non-INGRES data repositories all combine to form a single integrated logical database.

Gateways provide a means of integrating a variety of incompatible data repositories with a unified application development environment. We have already seen how INGRES/NET helps shield the user from different hardware platforms and networking protocols. INGRES/STAR increases the transparency by shielding the user from the location of data. Gateways shield the user from having to know the type of repository the data are stored in.

The advantages of gateways are several. First, the INGRES tool set can be used for rapid prototyping of applications. A programmer might use ABF instead of the traditional COBOL to prepare a series of reports and data browsers on an IMS database. The prototype could then be used to gain approval from management and users. At the conclusion of the prototyping phase, a more traditional program could be constructed using COBOL (or development could continue using ABF).

Another use of the gateway is the quick fix. QBF can be used to rapidly update data instead of writing a program to perform the operation. RBF or the Report Writer can be used to quickly generate reports on an ad hoc basis. In this case the INGRES tool set is supplementing the traditional programs.

Gateways are also a valuable tool for providing a transition into a relational environment. Moving a database from a hierarchical model to a relational model involves two sets of transitions. First, the data have to be moved from the hierarchical data model to the relational model. Second, the applications have to be rewritten to work in the new environment. With a gateway, the users can work on rewriting the application using the new tool set. The gateway allows the same application to be run against a relational database as a hierarchical one. Copying the data to the new environment then simply becomes a matter of copying the tables from one database to the other. Since both targets look like INGRES databases to the gateway user, a simple set of commands can be used to move the data.

Finally, gateways are useful in a production environment. The INGRES tool set can be used to perform sophisticated application development while maintaining current production. DEC, for example, markets the INGRES tool kit as a sophisticated application development environment for Rdb databases. The INGRES tools are used to supplement the traditional DEC user interfaces such as Rally and Teamdata.

For a distributed database environment, gateways are especially valuable. It is a fact of life that different areas of an organization will acquire different types of computer equipment and DBMS. The gateway allows an application to treat the heterogeneous data managers as a single logical database.

Since INGRES gateways use the services of GCF, it is possible for the different data repositories to be located throughout a heterogeneous network. With high-speed networks such as ISDN and FDDI, access to data becomes efficient as well as transparent. It would thus be possible to integrate several different databases into a single management information system:

- a DB2 database located at corporate headquarters
- an IMS database located at corporate MIS
- Rdb databases located at the manufacturing locations
- INGRES databases used for administration, personnel, and development

With INGRES/NET, it is entirely possible for all these systems to be located in different locations, even different states or countries. The INGRES/STAR distributed query optimizer makes sure that the relative bandwidths of the different data links are taken into account when processing distributed queries. To the application developer or the user, the location and type of data repository becomes irrelevant.

SQL Gateways

An SQL gateway allows an INGRES front end, including INGRES/STAR, to access another relational database system such as DEC's Rdb or IBM's DB2. The responsibility of the SQL gateway is to look like an INGRES database. The gateway modifies incoming SQL requests into a format compatible with the target system and modifies the data returned into a form compatible with INGRES.

The gateway has several important responsibilities. To the INGRES environment, the gateway must look like any INGRES database server. To the foreign data manager, the gateway must look like a normal user of that system. The gateway thus forms a buffer between the two environments. The responsibilities of the gateway include:

- accepting messages in the GCA format and reformatting them for the communication facility of the target system
- accepting INGRES standard SQL and modifying it into the appropriate dialect on the target system
- interfacing with the target system to execute the query
- translating all data returned into standard INGRES data types
- translating all error messages returned into standard INGRES error messages
- providing an interrupt mechanism to abort queries on the target system
- showing the native system catalogs as INGRES standard system catalogs

There are several things that the gateway is not responsible for. Most importantly, the gateway is not a two-way mechanism. There is no requirement that a DB2 program, for example, be able to access an INGRES server. Another function the gateway does not provide is to add to the functionality of the target system. If the target system does not have two-phase commit protocol or nested query capabilities, for example, the gateway will usually not provide that service.

If there is a function in the target system that is not available, it is the responsibility of the gateway to gracefully reject incompatible requests that are received. For example, it is possible that the target system only allows read-only access to data. The gateway must then reject any update requests by using a standard error message.

In order for INGRES tools and the distributed query optimizer to work properly, the gateway provides a series of standard catalogs that are queried for standard schema information. When the user of the gateway asks about the presence of an index, for example, the gateway provides a system catalog that describes available indices. This is usually done by providing a series of views over the native system catalogs.

All of the front-end systems, such as QBF, use a subset of SQL known as “Common SQL.” Common SQL consists of a subset of the various dialects used on most of the popular commercial systems, such as Rdb and DB2. All the INGRES front-end tools, such as QBF, have been written so that they generate Common SQL, accept standard error messages, and query the standard system catalogs. These tools work just as effectively on an Rdb database as they do on an INGRES database. Applications developed with INGRES 4GL or Embedded 4GL also work just as effectively with other systems using the gateways.

Each database accessed via a gateway looks like a single INGRES database to the user (see Fig. 9-1). With INGRES/STAR, these databases can be combined to form a distributed database. The *register* command can be used to make tables in the various components made known to the coordinator database.

Non-SQL Gateways

A non-SQL gateway allows the contents of nonrelational DBMSs to be included as tables in an INGRES database. Conceptually, data in a nonrelational system are treated as a simple heap or keyed file, equivalent to an INGRES table using a storage structure such as heaps, BTREE or ISAM. The non-SQL gateways in INGRES are just an extension of the access methods used by DMF (see Fig. 9-2).

Because the non-SQL gateway is implemented at the very lowest level of INGRES, issues like query languages are irrelevant. A query is parsed, compiled, optimized, and finally, DMF is asked to return certain records of data. Because the non-SQL gateway is implemented at the lowest level, integrities, permits, views, and optimizer statistics are all available on these non-INGRES objects. Native indices on the non-relational data are also supported.

The responsibilities of the non-SQL gateway are simply to perform DMF operations and possibly to participate in a recovery, rollback, or commit operation. It is possible that the gateway might be read-only environment, meaning that recovery support is not

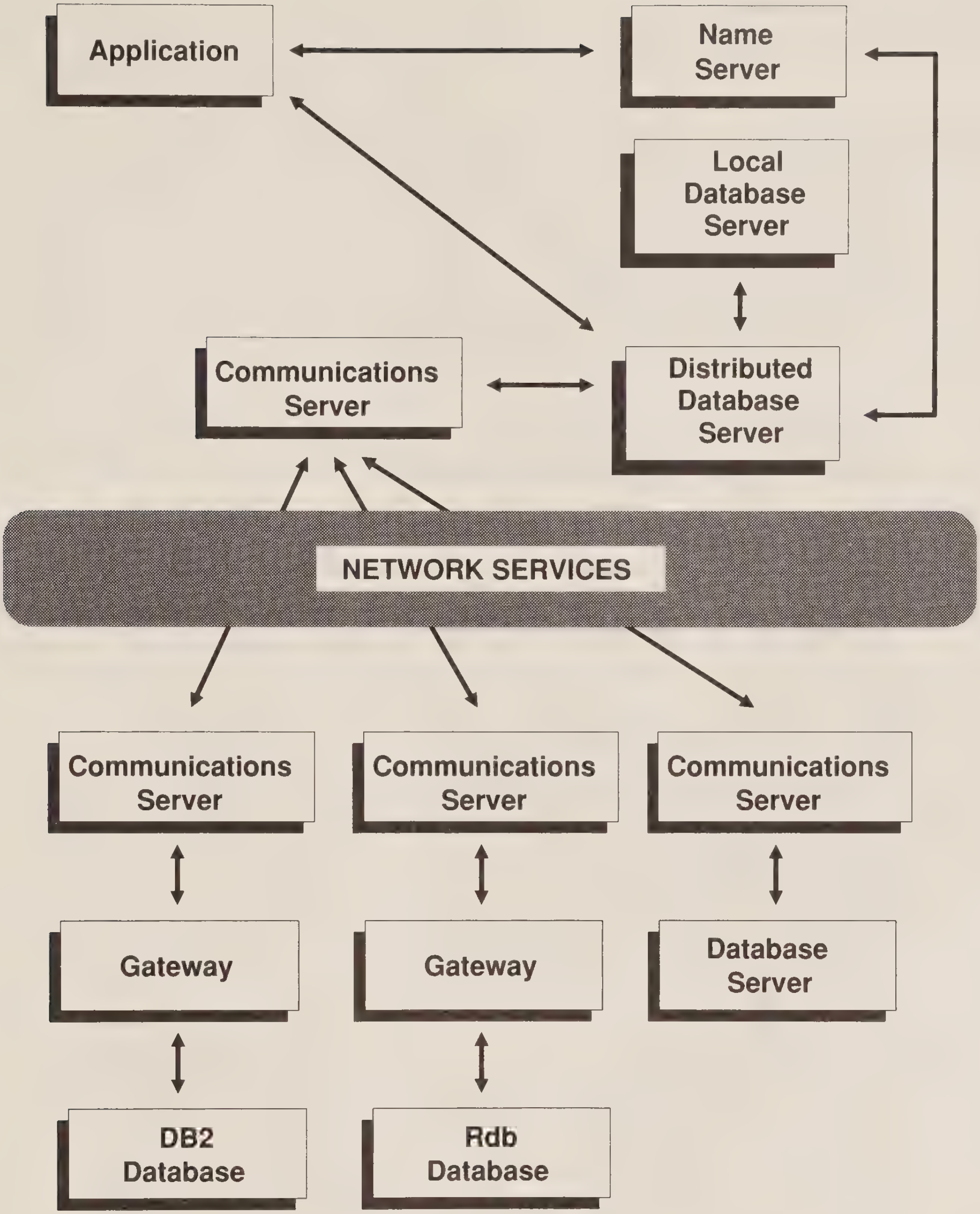


Fig. 9-1 SQL-Based Gateways (INGRES/STAR Example)

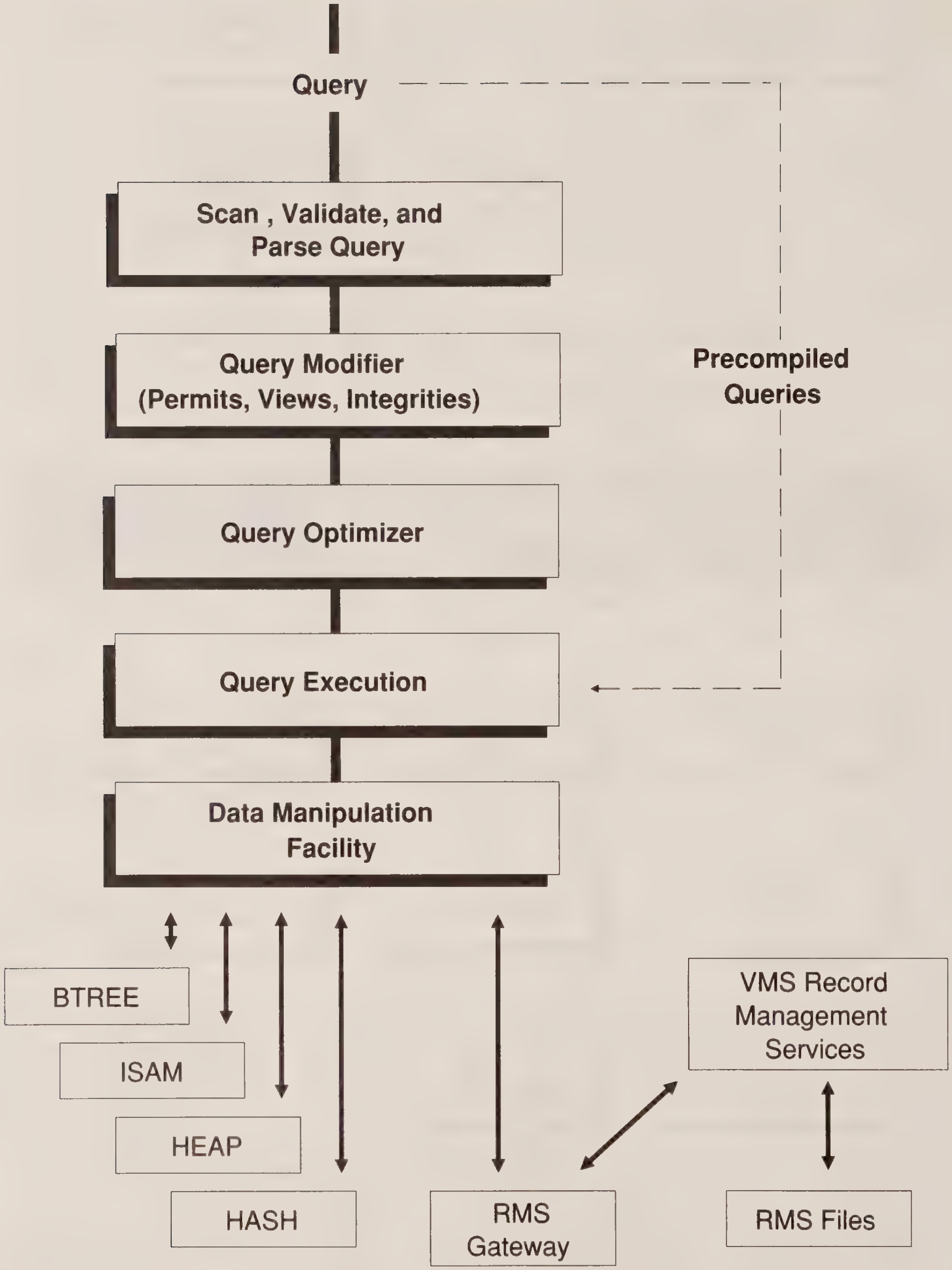

Query
Scan , Validate, and Parse Query
Query Modifier (Permits, Views, Integrities)
Query Optimizer
Query Execution
Data Manipulation Facility
Precompiled Queries
BTREE
ISAM
HEAP
HASH
RMS Gateway
VMS Record Management Services
RMS Files

Fig. 9-2 Non-SQL Gateways (RMS Example)

needed. Examples of non-SQL gateways are access to RMS files on a VAX running VMS, or access to VSAM files on IBM systems. Other examples of non-SQL gateways are IMS or IDMS/R databases.

The most important aspect of the non-SQL gateway is that we can take advantage of all of the INGRES facilities in this environment. Most important is the INGRES query optimizer, which can decide in what order to get data and whether or not to use secondary indices. Two specific non-SQL gateways are discussed to illustrate some of the issues that these data management systems can raise. First, access to RMS files shows some of the aspects of traditional file systems. Next, the IMS database is discussed to show how a hierarchical data structure is mapped to the relational model. Finally, some other potential applications of gateways are discussed.

RMS Files

The Record Management Services (RMS) is the file system on VAXs provided with the VMS operating system. RMS allows a wide variety of different files to be created by programmers. RMS files are also frequently used by the operating system itself to maintain information such as accounting data.

The RMS gateway allows these files to be defined to INGRES as tables. For example, the VMS accounting data could be defined as a table. Other information, such as user descriptions, could be maintained as normal INGRES tables. Reports could then be generated that joined the two tables together to provide bills and utilization reports. The RMS file is defined to INGRES using the *register table* command, which gives information to INGRES that is used to update the system catalogs. When a query is received and the target is an RMS file, the query optimizer treats the query just as it would any other data structure. Only a portion of DMF is aware that the origin of the data is a non-INGRES file.

The basic register table command might be as follows:

```
REGISTER TABLE ACCOUNTS ( sessionid = vchar(12),
                      usernameid = vchar(12),
                      start_time = vchar(25),
                      end_time = vchar(25),
                      cpu_usage = vchar(12) )
      AS IMPORT
      FROM 'sys$system:[sysmgr]accounting.dat',
      WITH DBMS = RMS
```

In this example, different parts of the file have been defined as different columns in an INGRES database table. Note that for purposes of illustration, the structure of the accounting.dat file has been simplified. The RMS file now appears to the user of an INGRES database as a table. SQL-based queries can be used to retrieve information from the table.

Several other options are available on the register table command. First, if the underlying RMS file is an indexed file, the key structure of that file can be declared to INGRES. This allows the INGRES query optimizer to use keyed access for the file if that will be more efficient then accessing all the underlying data.

There are three types of keyed structures. A "keyed" storage structure is equivalent to the ISAM table structure in INGRES. This allows range searches for data. The query optimizer knows, however, that data might be retrieved in unsorted order. The "sortkeyed" storage structure is equivalent to the INGRES BTREE, and data will be returned in sorted order. The "fullkey" storage structure is equivalent to an INGRES hash table. Only when all elements of the key are provided will keyed access be used.

If a structure is any form of keyed access, the KEY columns need to be declared to INGRES. It is also possible to declare the key columns to be in descending order instead of the default ascending order. Since each of the keyed storage structures is equivalent to an INGRES storage structure, the query optimizer makes the same types of decisions as it normally would, and can be unaware of the actual location of the data.

It is also possible to declare the number of rows in the underlying table. This information is used by the query optimizer to decide how to access the table. If no row count is given, the query optimizer assumes that there are 1000 rows in the table. A more accurate number allows the optimizer to make a better estimate of the number of data pages that will be returned by a particular query.

Another method to help the query optimizer is to run *optimizedb* on the RMS file. Since the file looks like an INGRES table, optimizedb runs just like it would on any other table. The statistics tables are then updated and this information is used to build more precise query plans.

There are four other options that can be specified using the register table command:

- recovery
- updates
- duplicates
- journaling

If the recovery option is specified, INGRES has full control over this object and can try to perform recovery in the case of aborted transactions or a system crash. If INGRES does not have exclusive control, trying to perform the reverse of an operation might not work. This is because some other entity may have tried to perform a subsequent operation on the same record.

Not allowing updates is one solution to the problem of conflicting access to a file. Often, INGRES is only needed for retrievals on the data. It would not make sense for an INGRES user to update the accounting.dat file on a VAX, for example. If update with recovery is needed on an object and exclusive control over the object is not available, a solution is to create a copy of the table. The user simply creates a new table from the RMS file as follows:

```
create table account_copy as
        select * from accounts
```

Note that the new table will not reflect any changes in the accounting file after the table is created.

Journaling and duplicates are two normal INGRES options for tables. Journaling tells the archiver in the data manager that it should save all transactions on this object into the journal files instead of discarding them. The duplicates option is a form of integrity protection that signifies that no duplicate key values are allowed.

If the file has a secondary index (in addition to the primary key), the *register index* command is used. Note that it is not possible to create a secondary index on an RMS file from within INGRES, because a secondary index has a series of pointers to pages in the primary table. Since the primary table is not managed directly by INGRES, INGRES cannot know what pages the data reside on and a secondary index would not be helpful. If the file system has a secondary index capability, however, INGRES is able to take advantage of that feature.

Most standard INGRES SQL commands work on an RMS file just as they would on any other INGRES table. Integrities and permits can be defined on the object. Of course, INGRES has no control over updates that are not made through INGRES. The integrities and permits only apply to INGRES-based access.

The only commands that don't work are those that change the physical attributes of a file. For example, the modify command will not work on an RMS file as it is used to change the storage structure of a table. If a different storage structure is needed, the developer has two options. First, she could use the services of RMS to create a new file with a different RMS storage structure. Second, she could create a new INGRES native table and modify that table to the new storage structure.

In the case of a file with only one type of records, importing data is a fairly straightforward proposition. However, in many file-oriented data systems, programmers have put various types of records into a single file. The application program was then responsible for decoding that information. To deal with this type of solution, the user can create views on the base table. First, the table is imported and given an INGRES table name. Then, views are created that specify how the data are interpreted:

```
CREATE VIEW record1 as
        SELECT * FROM base WHERE
                type = 1
                WITH CHECK OPTION
```

The check option makes sure that all data that is accessed or updated via this view meet the conditions in the where clause. A user could not update the view *record1* and specify a value of 2 for *type*.

Creating views to deal with multiple record types is also useful when a file contains header or trailer information. Often, most of the records in a file are of the same type, but there is some descriptive information at the beginning or end of the file. Creating a view with a check option is one way of eliminating this information from the retrieval.

IBM's IMS

IMS is IBM's hierarchical DBMS for the MVS operating system. Although many IBM sites are migrating to DB2, there are a large number of production applications still based on IMS. The IMS Gateway illustrates some of the issues in mapping hierarchical or network model database systems into a relational environment.

A hierarchical system structures all data in ordered segments. To access a lower, or child, segment, the upper or parent segment is first accessed. For each record in the parent segment, there are a series of rows in the child segment. Each child might have its own child segments, and so on down the hierarchy.

A simple hierarchy might be a membership roster for organizations. The organization name would be the parent segment, with departments as a child segment. A further, lower child segment to department would be employees. Each segment consists of pointers to data associated with the key value at that segment and pointers to child values. The organization segment key might be organization name. Associated with organization name would be the address of the organization.

Each organization name would then point to a series of department names. Department name would be the key to this next lower-level segment. Associated with the key values would be information about the department. Each department name key value would then point to a series of employee names.

The problem with a hierarchical database is that the programmer is required to navigate the hierarchy. Retrieving all information about the employee named "Martin" is not possible without knowing the organization and department that Martin works for. If the information is not known, the application programmer must search every organization and all associated departments and employees until the desired record is found.

Because the user must be aware of the physical structure of the database, IMS is in many ways similar to traditional file systems. Low-level commands to access data and the lack of SQL make IMS a candidate for the INGRES non-SQL gateways. The INGRES IMS gateway is similar to the RMS gateway in that various portions of the IMS database are defined as tables. The IMS gateway has the added complication of defining the hierarchy.

To import hierarchical data, such as an IMS database, into INGRES, it is necessary to import several different tables, each one corresponding to a separate level of the hierarchy. First, the top-level segment is imported as a table:

```
REGISTER TABLE parent (
                    pkey1 INTEGER,
                                       pkey2  INTEGER,
                                      data CHAR )
        AS IMPORT
        FROM database.segmenta,
        KEY = ( pkey1, pkey2 ),
        STRUCTURE = SORTKEYED,
        WITH DBMS = IMS
```

The root segment now looks like a table to the INGRES database, with two key values. If the user knows the key values, the IMS gateway can directly access the associated data. Otherwise, a search of all parent segments is performed until the correct one is found.

Next a child table is imported as follows:

```
        REGISTER TABLE child (akey1 INTEGER IS (parent.pkey1),
                              akey2 INTEGER IS (parent.pkey2),
                              ckey1 INTEGER,
                              data CHAR )
                   FROM database.segmentc,
                   KEY = (akey1, akey2, ckey1 )
                   STRUCTURE = SORTKEYED,
                   WITH DBMS = IMS
```

In this example, we have previously defined the parent table corresponding to the upper-level segment of the IMS database. The external format indicator on the column definitions tells the gateway that it should look for a table called "parent" with columns "pkey1" and "pkey2" when it is navigating the structure.

The user can treat the table "child" as a single INGRES table. The gateway will take all requests for data in the child structure and perform the appropriate IMS operation. If all key values are defined, the IMS operation will consist of directly retrieving the required segment. If the parent key values are missing from the query, the IMS operation will consist of a scan of all parent segments and the associated child segments until the desired data are found.

Updates to hierarchy keys are not supported in the INGRES IMS gateway. Updating a key would require all lower-level records to be moved. This limitation is not very severe, because most hierarchical database systems are designed so that key values rarely change.

Recovery in the case of IMS is a little more complicated than in the RMS example. In the case of RMS, the INGRES recovery manager simply rolls back the desired transaction. In the case of IMS, INGRES needs to coordinate with the IMS database manager. Both INGRES and IMS have similar utilities to ensure the integrity of a database, including checkpoints of a database and online logs of transactions. If a transaction aborts, INGRES simply sends a command to the IMS database system to roll back the relevant portion of the transaction. INGRES also rolls back any INGRES portions of the transactions. Recovery from aborted transactions is thus automatic in most situations.

Other Gateway Applications

Non-SQL gateways open a wide range of applications previously unavailable in a relational database environment. Any object that can be represented as a flat or keyed file can be accessible to a relational database using the gateway. For example, in a factory, a gauge on a vat could be equivalent to a row in a file. A programmable controller could periodically read the value on a gauge and move the data to a mailbox or socket on a computer. The mailbox or socket could then be registered as a table.

The application programmer would then write a program that selected all data from the registered table. Since INGRES retrievals do not timeout unless the user explicitly sets a parameter to do so, the select statement would wait until data arrived and then deliver them to the application. Once the reading on the gauge was retrieved, the program could perform a series of queries on other database tables to determine if the manufacturing operation is performing successfully. If an adjustment is needed, the program could write a new value to the table (another mailbox or socket in memory). That portion of memory would be monitored by a programmable controller that would accept the new value for the gauge and adjust the vat.

Another possible application of non-SQL gateways would be in a building surveillance application. Monitoring equipment could be configured so that movement or alarm indicators are mapped to files or portions of memory on a computer. The files are then registered as INGRES tables. When an indicator is set off, the application would then check other tables to see if there are supposed to be people in that portion of the building at that time. If not, a terminal at the security station can display a message that a certain area of the building should be examined.

To help users with these types of applications, a nonrelational gateway tool kit is available. The tool kit allows users to provide their own gateways for customized applications. Any data structure that can be mapped to a series of rows and columns can be accessible to an INGRES application using a gateway.

Heterogeneous Front Ends

The previous section discussed how INGRES front ends were able to access a variety of different data repositories. Many other user interfaces are able to effectively access the services of the INGRES database manager. This section describes a few of

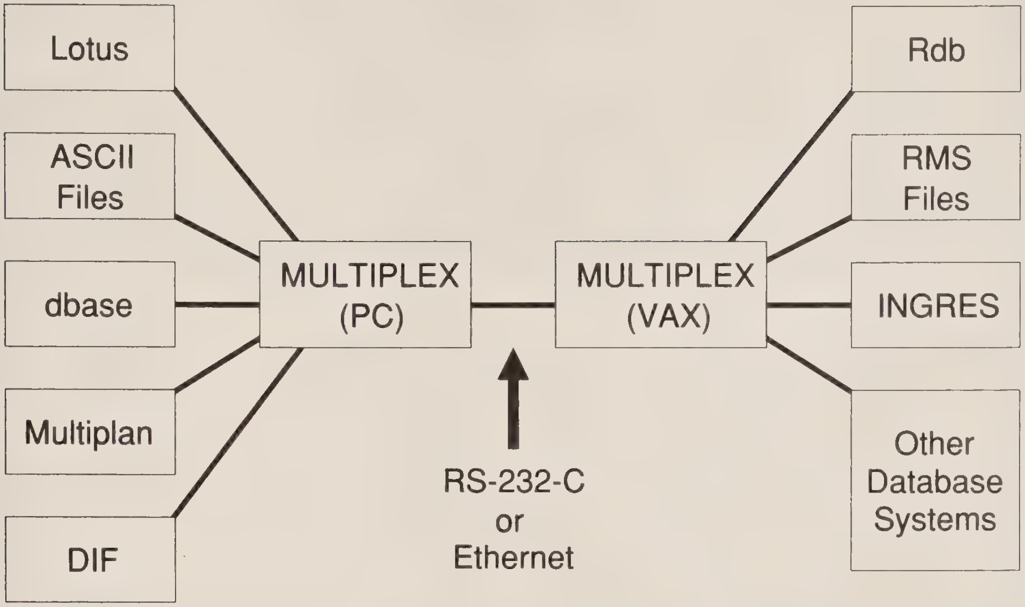


Fig. 9-3 Multiplex PC-Minicomputer Gateway

those front-end environments. It is fairly simple for vendors or users to port other applications to make use of the INGRES database services. Embedded SQL programs can dynamically construct SQL statements and dispatch them to the data manager. The results are then received and reformatted as needed by the application.

PC Access

Access to data from a PC can be accomplished in a variety of ways. One possibility is to run INGRES on the PC. This can be a combination of the front and back ends, just like INGRES on a minicomputer or mainframe. Alternatively, the user can run the front-end systems on the PC and use the services of INGRES/NET to access data managers and distributed databases located throughout the network. This configuration works just like any other version of INGRES, subject to the memory limitations of the PC environment.

In many cases, however, PC users insist on using the applications they are familiar with, such as Lotus 1-2-3. These tools may be more appropriate for the PC environment, especially if the user has spent a great deal of timing learning how to use those applications. Multiplex, developed by Network Innovations, is one solution to this type of problem. Multiplex provides a bridge between VAX-based data repositories and the PC environment (see Fig. 9-3). Users have a standard user interface that accesses a variety of data repositories on a VAX. These repositories include INGRES, as well as most other relational database systems in that environment.

Once data have been retrieved, they can be automatically converted into a format compatible with a variety of different PC-based tools. If Lotus 1-2-3 is picked, for example, the data are automatically formulated into a Lotus .WKS file. Then, the user can exit to Lotus, which will automatically retrieve the data into spreadsheet format. Data can also be moved into word processors, such as Multimate. The Data Interchange Format (DIF) is another format supported by several different vendors for loading in data.

Multiplex consists of two programs, one on the PC and the other on the VAX. The program on the VAX is the interface to the various data managers. The program on the PC is the user interface and prepares a query to send to the VAX program, just as QBF would prepare SQL and send it to a data manager.

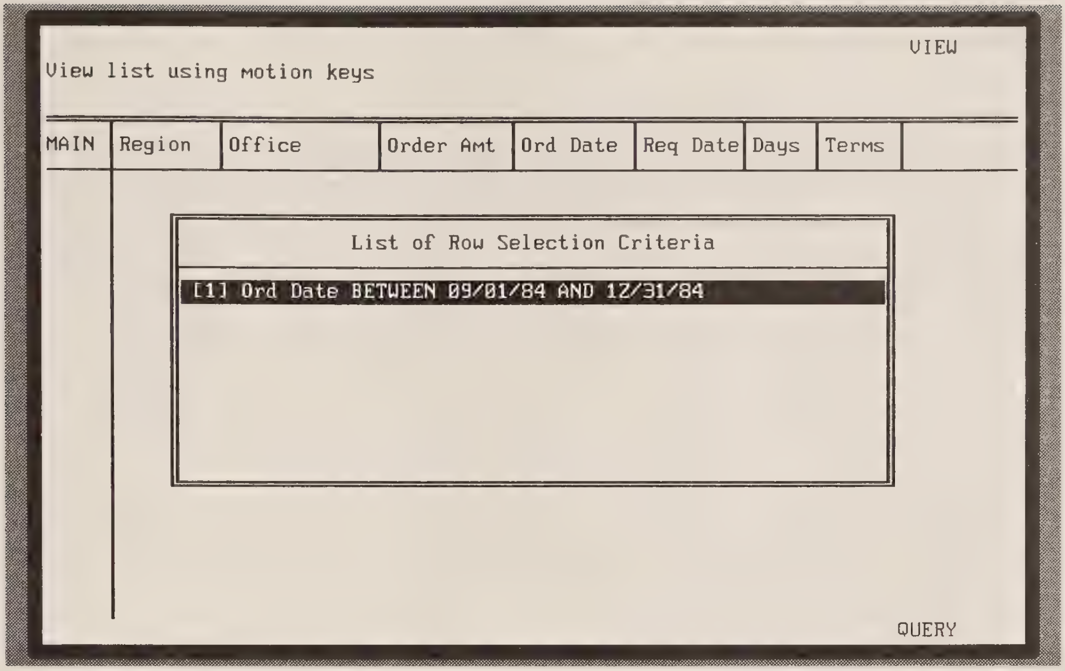
PCs or VAXs can be connected using a serial communications line or an Ethernet. Ethernet is more expensive, in that a controller card must be purchased for the PC. It does have the significant advantage, however, that the communications bandwidth is 10 mbps instead of the 1200 to 9600 bps of a serial line.

Defining a query to any of these target systems is done in a visual manner. The user picks a query target, such as a table and then establishes selection criteria on the table. For example, Figure 9-4 shows a restriction on a query that only selects rows where the order data are between 9/1/84 and 12/31/84. To establish a restriction, the user picks from a series of available menu options.

Once the query is defined, the user can download the data and browse the results (see Fig. 9-5). The query can be refined, such as sorting the data or eliminating certain columns. Once the proper data are defined, they can be converted into the proper format. This approach has both advantages and disadvantages. The prime advantage is that users are able to access central repositories and convert the data into a format they are familiar with. The tools that the user knows can be used for further formatting or analysis.

One disadvantage of this approach is that data can be retrieved from the database, but not put back in. Moving data back into the database has several implications on the integrity of data in that database. For that reason, most database administrators would not support the import of data into the back-end systems.

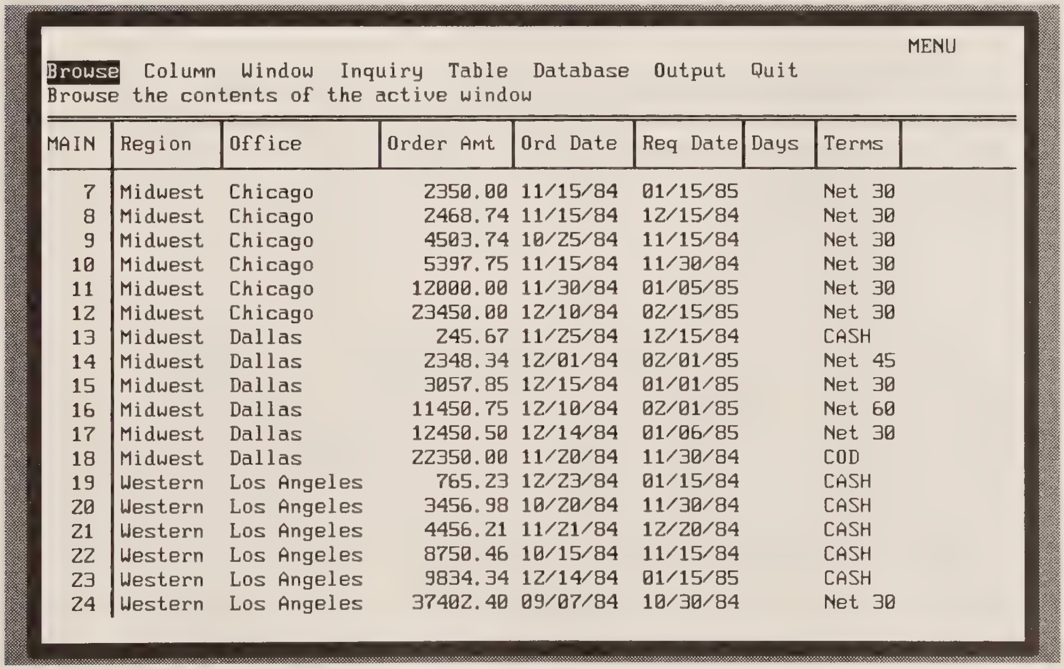
Once data leave the data repository, they in effect become a private database. The data are not accessible to other users and are not subject to the integrity constraints that a database imposes on the data. This is perfectly adequate for ad hoc analysis, but it leads to problems when a user starts changing data and the changes do not become available to other users.



```
View list using motion keys

MAIN | Region | Office | Order Amt | Ord Date | Req Date | Days | Terms

        List of Row Selection Criteria

  [1] Ord Date BETWEEN 09/01/84 AND 12/31/84
```

Courtesy of Network Innovations

Fig. 9-4 Selecting Data in Multiplex



MENU
Browse Column Window Inquiry Table Database Output Quit

```
Browse the contents of the active window

MAIN | Region  | Office      | Order Amt | Ord Date | Req Date | Days | Terms

   7 | Midwest  Chicago         2350.00 11/15/84  01/15/85         Net 30
   8 | Midwest  Chicago         2468.74 11/15/84  12/15/84         Net 30
   9 | Midwest  Chicago         4503.74 10/25/84  11/15/84         Net 30
  10 | Midwest  Chicago         5397.75 11/15/84  11/30/84         Net 30
  11 | Midwest  Chicago        12000.00 11/30/84  01/05/85         Net 30
  12 | Midwest  Chicago        23450.00 12/10/84  02/15/85         Net 30
  13 | Midwest  Dallas           245.67 11/25/84  12/15/84         CASH
  14 | Midwest  Dallas          2348.34 12/01/84  02/01/85         Net 45
  15 | Midwest  Dallas          3057.85 12/15/84  01/01/85         Net 30
  16 | Midwest  Dallas         11450.75 12/10/84  02/01/85         Net 60
  17 | Midwest  Dallas         12450.50 12/14/84  01/06/85         Net 30
  18 | Midwest  Dallas         22350.00 11/20/84  11/30/84         COD
  19 | Western  Los Angeles      765.23 12/23/84  01/15/84         CASH
  20 | Western  Los Angeles     3456.98 10/20/84  11/30/84         CASH
  21 | Western  Los Angeles     4456.21 11/21/84  12/20/84         CASH
  22 | Western  Los Angeles     8750.46 10/15/84  11/15/84         CASH
  23 | Western  Los Angeles     9834.34 12/14/84  01/15/85         CASH
  24 | Western  Los Angeles    37402.40 09/07/84  10/30/84         Net 30
```

Courtesy of Network Innovations

Fig. 9-5 Browsing Data in Multiplex

Macintosh Access

A related product, also developed by Network Innovations, is CL/1. CL/1 (for connectivity language) is language that allows Apple Macintosh systems to access data residing on a VAX. Instead of being a packaged user interface, CL/1 is instead a language that would be embedded into programs that run on the Macintosh. CL/1 offers a common language for programmers to access data on VAX systems as well as IBM VM and MVS systems. An application, such as a spreadsheet, could thus offer the user access to a variety of different data repositories located throughout the network.

The CL/1 language consists of four types of commands. First, the programmer can connect to a host data manager. Next, the programmer has a variety of SQL statements that can be used to retrieve data from the database into program variables. Third, program structure statements are used to test values of data retrieved, and perform looping and other control structures. Finally, output statements are used to send data back to the client application, which will then format the information for the user.

An example of a very simple CL/1 session would be:

```
open connection to Host "accounting" as user "john" password "X234" ;

open ingres dbms ;

open database "employee"

select emp_name, emp_sales, emp_quota from emp ;

if (emp_sales < emp_quota)

                call bad_employee() ;

close database;

close ingres dbms;

close connection;
```

This incomplete section of code would open a session with an INGRES database called “employee,” using the host “accounting” and the user name “John.” Next, three columns of data are retrieved from the emp table. If sales are less than quota, a procedure called “bad_employee” is called. Finally, the database connection is closed.

CL/1 has full support for the constructs in Embedded SQL programs. This allows the programmer to dynamically describe a database, tables, or columns. The user can then prepare a statement and have it executed. Multistatement transactions, updates to the database, and cursors are all supported.

CL/1 shields the programmer from variations in host access methods, networks, and particular brands of databases. Using the CL/1 language, the programmer always accesses data in the same fashion. The responsibility of CL/1 is to navigate the network, the host, and the particular brand of database to provide the data required.

Spreadsheets: 20/20

Just as users on a PC may wish to do their analysis in Lotus 1-2-3, VAX-based users may also wish to use a spreadsheet for doing analysis. 20/20, distributed by Access Technology, offers a Database Connection option that allows the user to access data in a variety of different data repositories. The database connection starts by opening up a particular target database. Next, the user can either enter direct SQL statements or use a forms-based interface to define the query (see Fig. 9-6). The forms-based interface allows the user to point to a particular table or view in the database. Next, he specifies sort and selection criteria for the retrieval. The user can browse the data to decide if the query is appropriate. When finished, the data are imported to a particular range in the spreadsheet.

Once in the spreadsheet, the user can perform all the various operations, such as summations or other formulas, that a spreadsheet supports. For example, in Figure 9-7 the user has added a total budget column for data retrieved from a projects table. The user has also specified formatting commands so that the budget column is displayed in a monetary format. Both the spreadsheet and the query can be saved. When the spreadsheet is reloaded, the user can have the query executed again. The data are refreshed and the various derived columns recalculated.

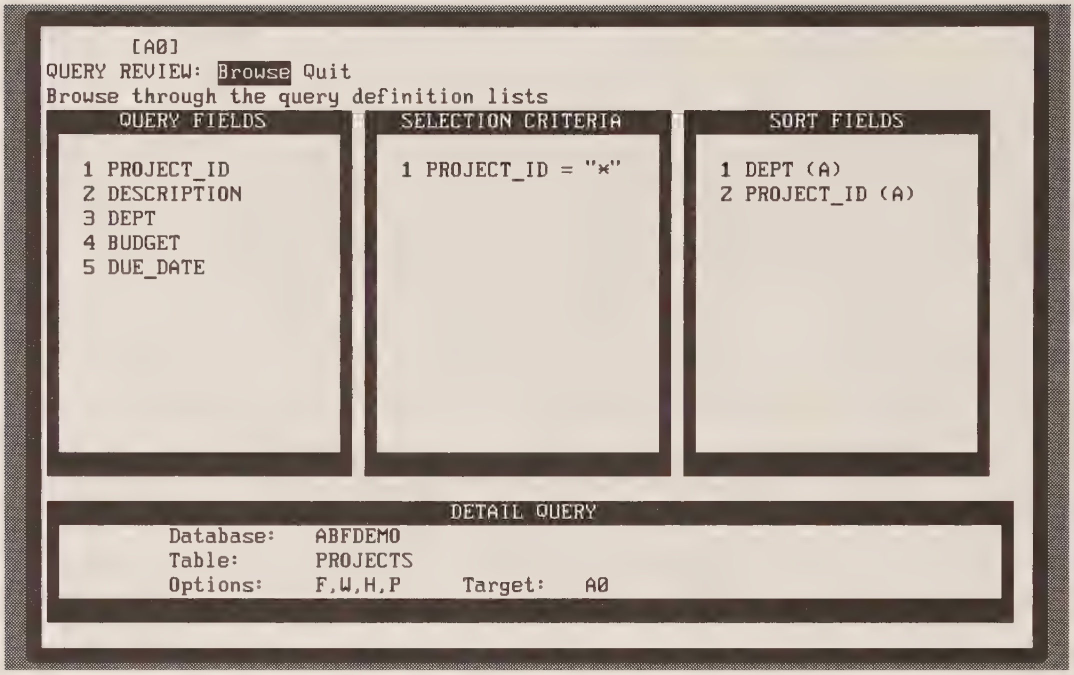
Natural Language Interfaces

The Natural Language Interface (NLI), a product of Natural Language Incorporated, allows a user to use English to query a database. NLI supports INGRES, as well as several other databases such as Oracle and SyBase. NLI is able to provide a distributed database capability by dispatching queries to different data managers and combining the results. Alternatively, NLI can use the services of INGRES/STAR to access a variety of data sources.

Tools like QBF provide a great deal of flexibility, but they require the user to work against established query targets. A new JoinDef can be constructed, but this requires the user to specify master/detail relationships and possibly update and delete rules.

NLI allows the user more flexibility for ad hoc queries. A powerful English language parser is able to respond to a large number of different formulations of queries. Because NLI also includes a series of rules that pertain to the database, it is able to evaluate semantic information. A user can tell NLI that "good" performance is where salespeople sell more than the previous years sales. Then, a user can ask the NLI "which salespeople are doing well?" NLI will be able to equate the semantic concept of well with the rule that shows a "good" salesman (see Fig 9-8).

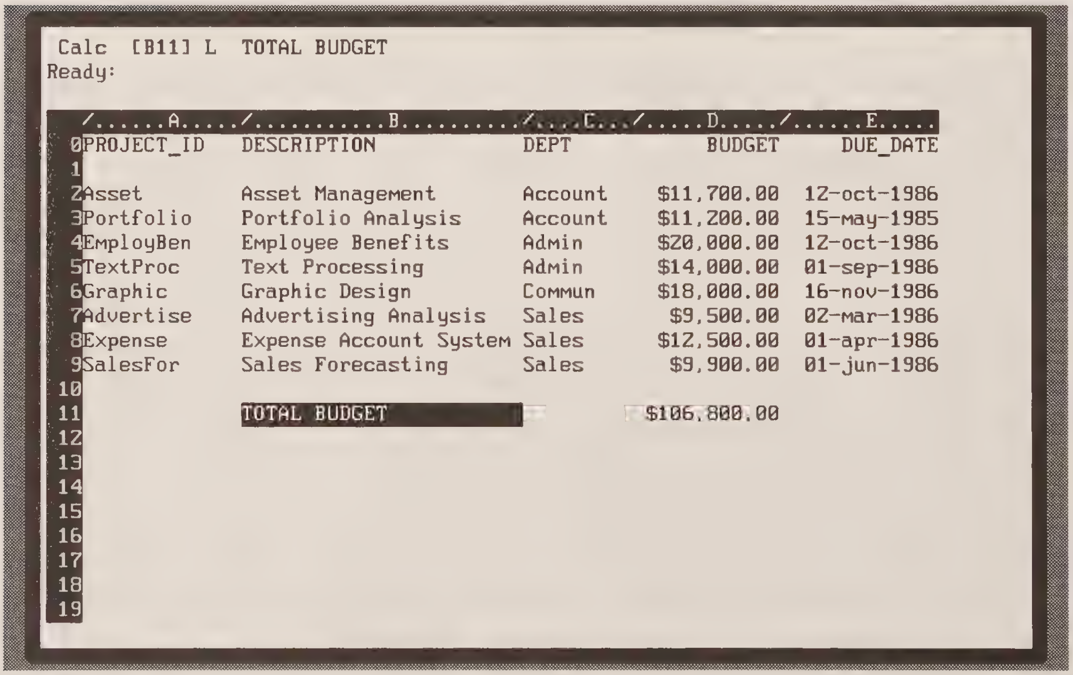
The English language interface thus has two important advantages over more traditional forms-based interfaces for ad hoc queries. First, the parser contains semantic information allowing the user to formulate imprecise queries and have them evaluated.



```
     [A0]
QUERY REVIEW: Browse Quit
Browse through the query definition lists
     QUERY FIELDS              SELECTION CRITERIA               SORT FIELDS

  1 PROJECT_ID             1 PROJECT_ID = "*"             1 DEPT (A)
  2 DESCRIPTION                                           2 PROJECT_ID (A)
  3 DEPT
  4 BUDGET
  5 DUE_DATE

                                   DETAIL QUERY
          Database:    ABFDEMO
          Table:       PROJECTS
          Options:     F,W,H,P      Target:    A0
```

Courtesy of Access Technology

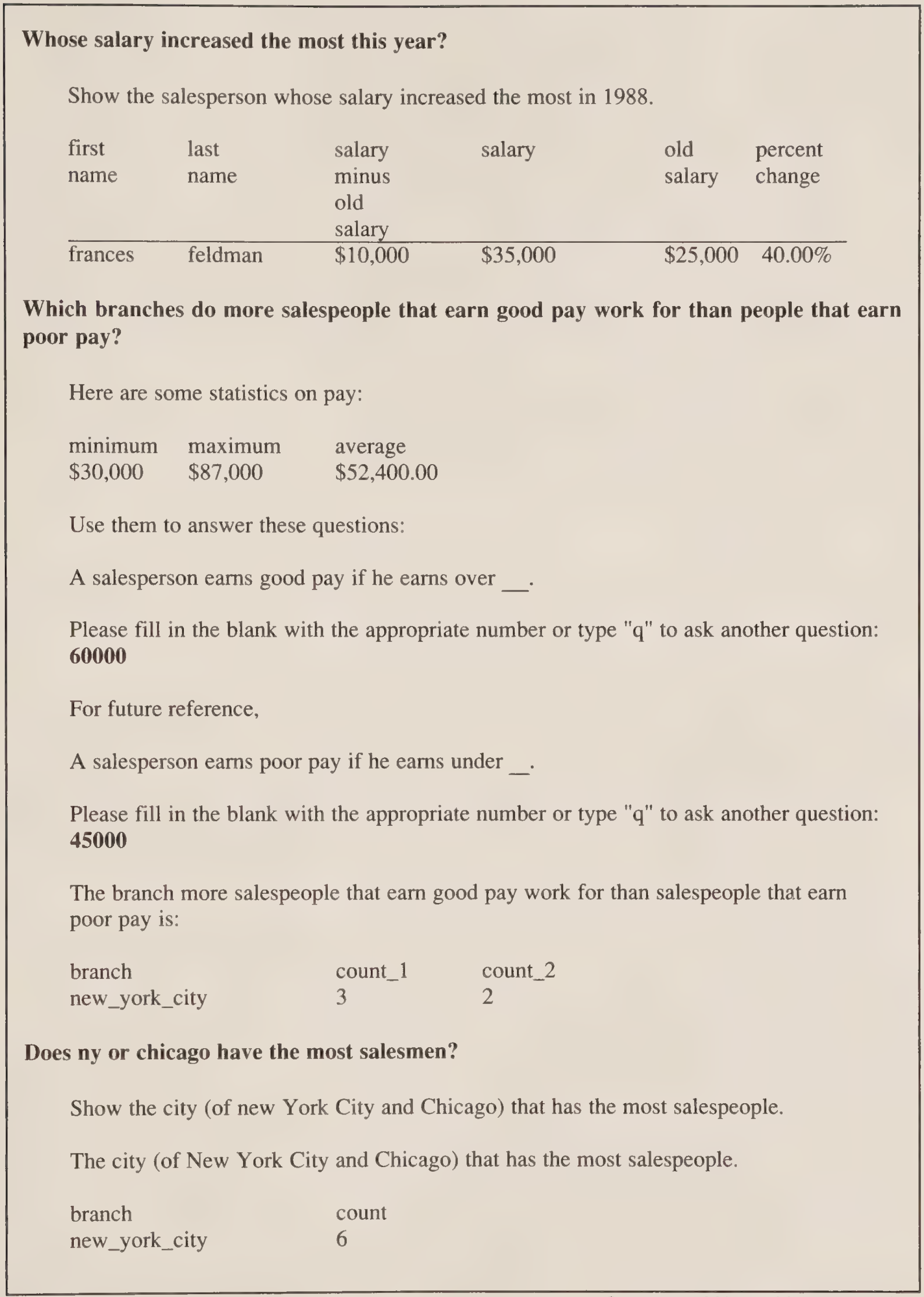
Fig. 9-6 20/20 Database Connection: Search Criteria



```
 Calc  [B11] L  TOTAL BUDGET
Ready:

   /......A...../............B........../....C.../.....D...../......E.....
  0PROJECT_ID   DESCRIPTION            DEPT            BUDGET      DUE_DATE
  1
  2Asset        Asset Management       Account     $11,700.00   12-oct-1986
  3Portfolio    Portfolio Analysis     Account     $11,200.00   15-may-1985
  4EmployBen    Employee Benefits      Admin       $20,000.00   12-oct-1986
  5TextProc     Text Processing        Admin       $14,000.00   01-sep-1986
  6Graphic      Graphic Design         Commun      $18,000.00   16-nov-1986
  7Advertise    Advertising Analysis   Sales        $9,500.00   02-mar-1986
  8Expense      Expense Account System Sales       $12,500.00   01-apr-1986
  9SalesFor     Sales Forecasting      Sales        $9,900.00   01-jun-1986
 10
 11             TOTAL BUDGET                      $106,800.00
 12
 13
 14
 15
 16
 17
 18
 19
```

Courtesy of Access Technology

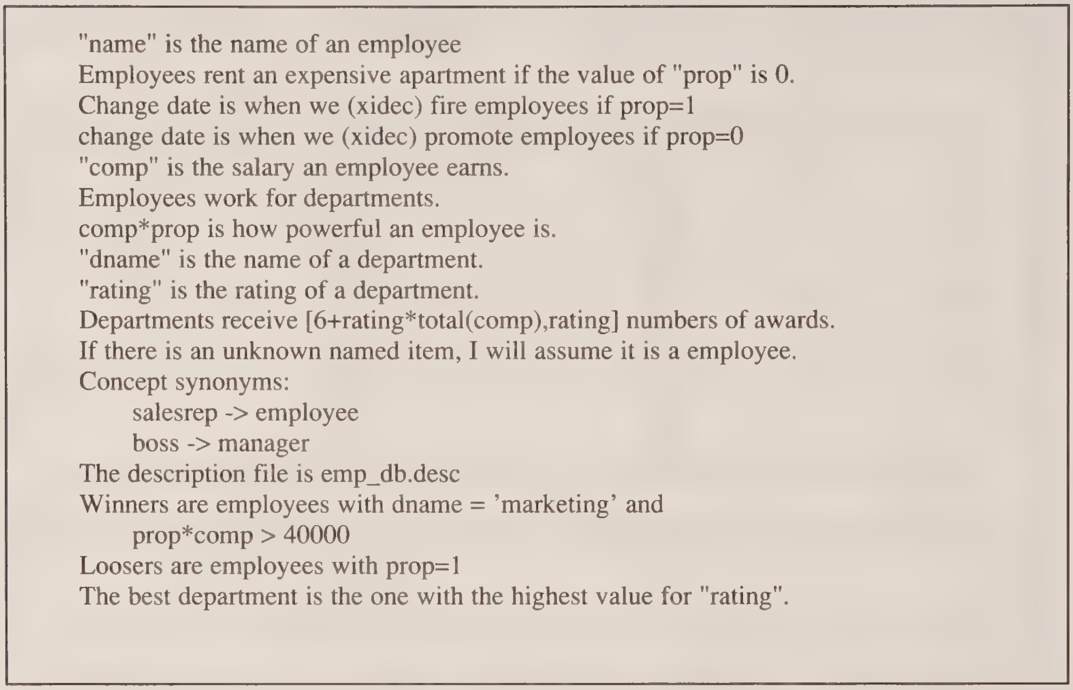
Fig. 9-7 20/20 Database Connection: Query Results



```
Whose salary increased the most this year?

    Show the salesperson whose salary increased the most in 1988.

    first      last       salary      salary        old       percent
    name       name       minus                     salary    change
                          old
                          salary
    ---------------------------------------------------------------------
    frances    feldman    $10,000     $35,000       $25,000   40.00%

Which branches do more salespeople that earn good pay work for than people that earn
poor pay?

    Here are some statistics on pay:

    minimum    maximum    average
    $30,000    $87,000    $52,400.00

    Use them to answer these questions:

    A salesperson earns good pay if he earns over __.

    Please fill in the blank with the appropriate number or type "q" to ask another question:
    60000

    For future reference,

    A salesperson earns poor pay if he earns under __.

    Please fill in the blank with the appropriate number or type "q" to ask another question:
    45000

    The branch more salespeople that earn good pay work for than salespeople that earn
    poor pay is:

    branch                count_1     count_2
    new_york_city         3           2

Does ny or chicago have the most salesmen?

    Show the city (of new York City and Chicago) that has the most salespeople.

    The city (of New York City and Chicago) that has the most salespeople.

    branch                count
    new_york_city         6
```

Courtesy of Natural Language, Inc.

Fig. 9-8 Session Using the Natural Language Interface



```
"name" is the name of an employee
Employees rent an expensive apartment if the value of "prop" is 0.
Change date is when we (xidec) fire employees if prop=1
change date is when we (xidec) promote employees if prop=0
"comp" is the salary an employee earns.
Employees work for departments.
comp*prop is how powerful an employee is.
"dname" is the name of a department.
"rating" is the rating of a department.
Departments receive [6+rating*total(comp),rating] numbers of awards.
If there is an unknown named item, I will assume it is a employee.
Concept synonyms:
    salesrep -> employee
    boss -> manager
The description file is emp_db.desc
Winners are employees with dname = 'marketing' and
    prop*comp > 40000
Loosers are employees with prop=1
The best department is the one with the highest value for "rating".
```

Courtesy of Natural Language, Inc.

Fig. 9-9 Description of an NLi Knowledge Domain

Second, the system is ad hoc. A user can look at various questions, examine the results, and pose a new question.

NLI includes several features that simplify the processing of formulating questions. First, if a word is misspelled by the user, NLI attempts to make a correction. Second, incomplete sentences can be included. If NLI can make some sense of the question, it repeats it back to the user and shows her the results. By echoing back the question, the parser shows the user how the question was interpreted. If that was not the desired information, the user can reformulate the question in a more precise manner. This approach contrasts to asking the user what he meant. The problem with trying to ask the user for a more precise definition is that the dialogue with the parser can continue indefinitely. The NLI approach tries to figure out what the user probably meant and shows him the answer to that question.

The NLI Connector is used to define a domain of knowledge that pertains to a particular database. The database administrator starts by defining the schema of a database in a file. Then, the connector asks a series of questions that help define the semantic content of the database. This helps describe what the information means. For example, the column “name” can be defined as being the name of an employee (see Fig. 9-9). Then, when the user asks for the names of all employees, the parser is able to formulate an SQL query that looks for the name column. The administrator can also formulate rules and categories that help define the information. For example, winners and losers can be defined.

The concepts of winners and losers are examples of subjective values that are equated to a particular type of query on the database. The administrator can also define other types of information, such as derived data. For example, the rewards that a department receives can be a function of sales performance and other factors. The result of this definition process is a domain of knowledge about a particular database or set of databases. Users can then increase this domain of knowledge by putting their own rules and definitions into the system. Over time, the domain is customized to a particular group of users and their terminology and concepts.

It is also possible to make use of NLI from within a programming environment. Take, for example, the process of defining parameters for a particular report on the database. One way to do this is to enter the parameters on a form and run the report. If the data are broad or too narrow, the parameters are reentered and the report is run again. An alternative would be to use NLI to define the parameters for the report. The programmer could include the following into the INGRES 4GL code:

```
'DEFINE' = {
                query_buf := call system 'NLI' ;
        }
```

The result of this call is a properly formatted SQL query. The programmer can then use that query to define a report to be run. Note that NLI also provides a form of reporting capability, including the ability to define the format of a report. This tool can then be used in conjunction with the INGRES Report Writer, with the Report Writer being used for more complex, highly formatted reports.

Links to Knowledge Bases

Several links exist between INGRES and artificial intelligence products such as IntelliCorp's KEE and Inference Corporation's ART. The KEEconnection interface to databases is discussed here for illustration. A product like KEE is used for developing knowledge bases and artificial intelligence applications. An expert system is an example of such an application. These knowledge bases, in order to be effective, need to be able to access data from a database where production information is stored.

A KEE knowledge base consists of a series of units. Each unit is equivalent to the concept of an object in the Picasso project discussed earlier. The unit has a series of slots. Slot values can be simple values or can contain other units. A facet for a slot is a description of the data that can go into it, a form of integrity constraint. A slot can also contain a method. A method is simply a series of steps that is carried out. When the steps are finished, a message is sent back to another unit. A special type of unit is the demon unit. A demon unit monitors the values in a slot. Whenever that slot is accessed or changed, a method is activated. The knowledge base can thus be active, responding to changes in the environment.

Units are organized into classes and subclasses. A subclass inherits the slots from a parent member. This allows subcategories to be described without redescribing the basic attributes of the category. For a knowledge base to access a database, there needs to be a mapping between the two data environments. The KEEconnection product is used to provide that mapping. This process begins by reading the schema of the target database and creating a default one-to-one mapping between database tables and units in a knowledge base.

The programmer can then change the default mapping using a graphic-based editor. Joins can be defined, for example. New types of slots can be created that hold the foreign key for a join. Mapping information is kept in a separate mapping knowledge base. When the application needs to access data, KEEconnection consults the mapping knowledge base to find the appropriate targets in the database. It then generates SQL queries. When the data are returned, KEEconnection again consults the mapping knowledge base and then creates appropriate units from the data and places them in the application knowledge base.

Note that once the mapping is defined, the application does not have to worry about how to get the data. It simply requests the information and it is retrieved from the database. The programmer can then analyze that information using a series of rules and other techniques. When a rule leads to another rule, that could also require data to make a decision. Eventually, a goal or answer is arrived at and presented to the user.

Products such as KEE and ART have many similarities to the Postgres project carried out at UC Berkeley. Note that Postgres also included a rule mechanism, procedures (methods), inheritance, and other attributes of object-oriented programming environments.

Access from Other Environments

A variety of other tools are available to access INGRES databases. Some of these are general-purpose tools, such as Resource, Inc.'s link between WordPerfect and INGRES. This tool allows data from the database to be placed into mail/merge applications that use WordPerfect files.

Other general-purpose interfaces exist to products such as BBN's RS/1 modeling environment. RS/1 is a software product used for scientific analysis. Instead of reading the data from a file, the RS/1 links allow the data to be stored in the database and take advantage of query optimization, concurrency control, and other database features.

Several other products exist that are customized for particular applications. These Value-Added Resellers (VARs) write applications for the particular needs of an industry or niche market, allowing customers to quickly use the application instead of designing a new one.

Summary

This chapter looked at two different kinds of integration. Gateways allow INGRES front ends to access a variety of different data repositories. The other type of integration allows other front-end systems to access INGRES data repositories.

Gateways allow SQL and non-SQL data managers to be accessed. The SQL gateway looks like an INGRES data manager to the front-end application. The responsibility of the SQL gateway is to accept INGRES queries and modify them into a format compatible with the back-end data management system. The front-end application can be a tool such as QBF or an INGRES/STAR process, which is in turn used by an INGRES application such as QBF. In both cases, INGRES/NET can be used to provide access to a heterogeneous network.

Non-SQL gateways integrate file systems and hierarchical databases and link them to INGRES tables. The INGRES query optimizer is then used to formulate a query plan, which is translated by DMF into low-level calls. The low-level calls can access native INGRES tables or can be routed to the non-INGRES data manager such as the IMS database system.

Foreign front ends are just another INGRES application. Instead of using QBF, the user uses another interface. There are a wide variety of such products, ranging from natural language and artificial intelligence to spreadsheets and word processors. Foreign front ends issue SQL, just as QBF or any other INGRES front-end application would. The advantage of these other interfaces is that they provide users with a wider range of options.

Part IV

Managing Development

Overview

The focus in the first three parts of this book has been on tools for accessing data. Little attention has been given to how an organization decides what data to keep or what applications to build. This last part of the book shifts the focus to the design process: how to design a database or application, how to communicate the design to users and developers, and how to structure the computing environment so that change does not make either software or data (or users) obsolete.

Meta-data are used to define the structure of applications and databases. The repository for meta-data in the INGRES environment is a special set of tables in a database called the data dictionary. Chapter 10 discusses the structure of a data dictionary and the different ways to access the information in it.

Computer-aided software engineering (CASE) uses the data dictionary to form a model of applications and databases. These modeling tools, discussed in Chapter 11, let the developer use graphic-based design methods to form a picture of a part of the information system and then help the developer turn that model into applications and databases.

Chapter 12 discusses how to select tools in a way that allows integrated access to data in a rapidly changing environment. The information architecture structures the tools in terms of a set of key interfaces between the tools, the data, the user, and the underlying network. A well-designed information architecture allows continued access to data, portability of applications, and the ability to change one component of the computing environment without restructuring all of the others.

Chapter

10

Data Dictionaries

Meta-Data and Data Dictionaries

Data dictionaries are simply meta-data: data about data. When a user types *select * from emp*, she is asking for data in the emp table of the database. When INGRES receives that query, it first goes to a set of tables in the database that stores meta-data, called the system catalogs. From the system catalogs, INGRES can find out if emp exists, what columns it has, and information used by the query optimizer such as the presence of statistics or particular storage structures. After querying the system catalogs and formulating a QEP, INGRES goes to the tables that actually contain the data to execute the query.

The definition of an INGRES database, including all front-end objects, is managed using the same relational model that users use to manage their data. The advantage of this approach is that database services, such as lock management, can be applied to the meta-data as well as to the data. Keeping meta-data in relational tables allows users to access meta-data using the same tools for accessing ordinary data. All of the power of SQL is available to manipulate the meta-data.

Data dictionaries are also used directly by users and application developers. When a user selects a table from a table field as a QBF query target, he is using the services of the data dictionary. This means that the user does not have to know in advance all the tables, or the particular spelling of a table's name in the database.

A more general form of data dictionary is used for CASE tools. The INGRES/team*work* software examined in the next chapter makes extensive use of a data dictionary for storing model elements. A model element, such as the definition of a particular function in an application, defined in one type of modeling environment, such as a systems design tool, is then available for use in other modeling tools.

A standard format for data dictionaries is the Information Resources Dictionary System (IRDS). IRDS has been adopted as a standard by the American National Standards Institute (ANSI) and is also being developed as an international standard by the International Organization for Standardization (ISO). IRDS defines a set of standard operations on meta-data. These standard operations, such as retrieving the definition of an element, allow a consistent method for accessing meta-data as well as allowing the migration of data definitions from one dictionary to another. One dictionary could be implemented as an Rdb database, and that information could be moved over to an INGRES environment without writing a program to do a conversion of the data.

General data dictionaries such as IRDS are extensible, meaning that users can add definitions of new types of data. A database has objects, such as tables, columns, and views, that are defined in the data dictionary. A user could add the concept of projects and tasks and then query the data dictionary to see which data repository contains which projects.

This chapter begins with a discussion of the INGRES data dictionary and how it is used by various INGRES subsystems. Then, the IRDS standard is presented. The chapter concludes with a discussion of some potential uses of data dictionaries.

The INGRES Data Dictionary

The INGRES data dictionary has two main purposes. First, the INGRES subsystems use the data dictionary to store and access the definitions of objects. These objects can be front-end objects, such as forms or QBF join definitions, or back-end objects, such as tables, indices, or permits. The data dictionary can also be directly used by people to find out what objects are available. For example, a user could look at a list of available reports to decide which report is appropriate for a particular application.

Data dictionary information is stored in tables. When a front or back end wishes to access data dictionary information, it simply issues an SQL statement. The back end takes the SQL statement and formulates a QEP. Since access to data dictionary information uses the same mechanism as ordinary data, all of the services of the back end are available for efficient access to information while preserving the integrity of that information. For example, since multiple users may be reading and writing data dictionary information, the back end can secure appropriate locks on tables to ensure that inconsistent operations are not performed.

The INGRES data dictionary has two levels. First, there is a special database called *IIDBDB* that keeps track of information that pertains to the entire INGRES installation, such as a list of valid INGRES users (see Fig. 10-1). Second, there is a set of tables in every database, called system catalogs, that keep track of all objects that make up that particular database.

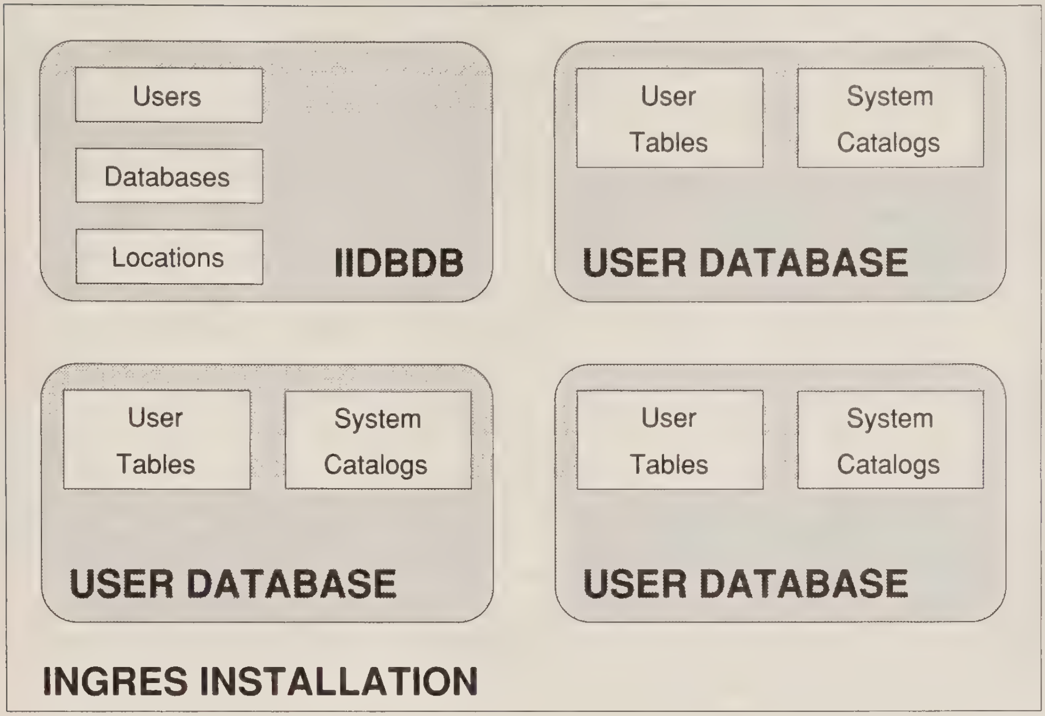


Fig. 10-1 Data Dictionaries in an INGRES Installation

The Database Database

The first level of data dictionary in the INGRES environment is the database database—IIDBDB. This is a database, like any other INGRES database. The IIDBDB is used to keep track of users, databases, and locations in a particular installation. IIDBDB is thus a meta-database and contains general information pertinent to all databases.

When a user starts up a front-end system and connects to a server, one of the first things that is done by the server is to consult IIDBDB to see if the user is an authorized INGRES user. Limiting access to databases to authorized INGRES users is the first line of security for access to data. When a server connects to a particular database, it also consults IIDBDB. The first thing the server does is check to see if the database actually exists. If the database exists, it then looks for the location names that the database uses. Location names point to particular devices—disk drives—on a computer. By checking the location names, the server knows where to find data for a particular database.

Two utilities are normally used to check on information in IIDBDB. The *accessdb* utility is only available to the INGRES superuser. This superuser is responsible for administering the INGRES environment on a particular computer. This is in contrast to the database administrator, who is the owner of a particular database and is responsible for administering that subenvironment.

Accessdb is used to validate users on the computer as valid INGRES users. Various permissions can be given to that user, such as allowing her to update system catalogs. Normally, users are not given that permission. Accessdb can also be used to give users superuser access. Superuser access allows the user to impersonate another user, including the database administrator, of any database on the system.

Catalogdb is a similar utility, used by any user. A user can examine what databases he owns and who has permission to access them. Catalogdb allows the user to look for a particular database or to examine a catalog of information. Figure 10-2 shows an example of the catalogdb utility. The screen displays the information on a particular user and any special privileges, such as superuser access, that the user has. The screen also shows the databases that the user owns, as well as databases owned by other users that she may access. Catalogdb and accessdb are applications that access IIDBDB. General-purpose tools such as QBF can also be used to query the database. A user simply runs QBF and specifies IIDBDB as the database name. Then, the user picks a query target, such as the table of valid INGRES users.

In addition, applications can be written that use the information in IIDBDB. For example, a user might wish to write a utility that generated a report of the amount of disk space used by each database at a particular installation. Since new databases can be created at any time, the utility would first check IIDBDB to see which databases exist. Then, the utility would use the location names for each database to find the files that make up the database and total the amount of disk space used.

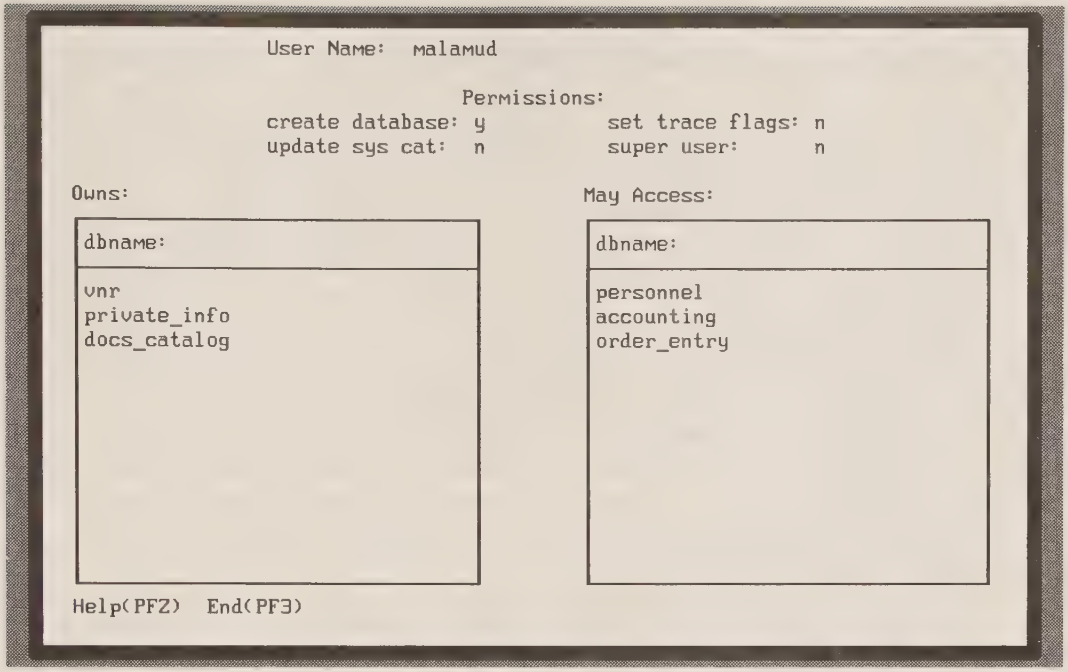
INGRES System Catalogs

Each database in INGRES has a set of system catalogs used to store meta-data about the particular database. Whenever a new database is created, these tables are all constructed. There are three types of system catalogs in an INGRES database:

- implementation-specific database tables
- a standard catalog interface to the database
- front-end catalogs

INGRES supports various types of database environments, ranging from full-fledged INGRES databases to gateway systems. Each of these different implementations has different tables that describe the data in that particular type of system. A normal INGRES database, for example, has a series of tables that describe storage structures and secondary indices used in INGRES. A gateway to Rdb contains information on the particular storage structures and indices used in Rdb databases.

In an INGRES environment, the front-end program should be unaware of the particular type of back end it is accessing. INGRES databases, gateways to non-INGRES sources, and distributed databases should all appear the same to the front-end application as well as to the user. To accomplish this level of transparency, INGRES defines a



```
                 User Name:  malamud

                            Permissions:
                 create database: y          set trace flags: n
                 update sys cat:  n          super user:      n

  Owns:                                     May Access:

  dbname:                                   dbname:

  vnr                                       personnel
  private_info                              accounting
  docs_catalog                              order_entry

  Help(PF2)  End(PF3)
```

Courtesy of Relational Technology

Fig. 10-2 Using CATALOGDB to Access IIDBDB Information

standard catalog interface that is used by front ends. The standard catalog information consists of information common to the various types of data repositories accessed by INGRES. Often, the standard catalogs are implemented as views on top of the implementation-specific system catalogs.

The *IITABLES* system catalog is an example of a standard catalog that lists all tables available in the database. Within the IITABLES table, there is information that pertains to every table, including the name of the table, the table owner, and the date that it was created and last altered. Two other columns specify what type of table is being referenced. A table can be a real table, a view, or an index. The table can also be a native INGRES table, a link to another database using INGRES/STAR, or an imported table using the non-SQL gateways.

If the table is a native INGRES table, the IITABLES catalog contains information on whether there are permits or integrities on the object. There is also a special column in this table that indicates if the "all to all" or "retrieve to all" permissions have been set. This means that the query processor can continue processing the query without consulting the *II_PERMITS* table.

The II_PERMITS catalog contains permission information for more complicated security restraints. The query modifier takes this information and adds it to the text of the query before passing it on to the query optimizer. The II_PERMITS table lists the object in question (such as a table), the user to which the permit applies, the owner of the object, and a text segment that has the SQL or QUEL for the permission itself.

The schema, or definition, of an INGRES database is entirely contained in the system catalogs. In addition to tables and permits, other tables contain information on integrities, columns, indices, and optimizer statistics. All of these tables are used by the data manager in servicing SQL requests, as well as directly by end users.

The front-end subsystems also use database tables to store definition information. These front-end tables are known as the extended system catalogs. Every object created by front ends is stored in these extended system catalogs, including VIFRED forms, QBF JoinDefs, RBF reports, and ABF applications. Each object is defined in the *II_OBJECTS* system catalog, which includes the name, type, and a unique id number for each object, as well as the owner. It also includes information global to all objects such as the creation date, the last alteration date, the number of times it was altered, and by who. The catalog also includes a short remark column used for brief descriptions.

The *II_LONGREMARKS* catalog is used for extended documentation on objects. Each of the front-end catalog utilities allows the user to enter extended remarks for objects, which are then stored in this table. The short and long remarks columns are shown in the catalogs displayed by each front end. For example, the reports subsystem displays a list of all available reports and the short remark. A menu option lets the user examine the long remark for the report that the cursor is on.

In addition to the global front-end system catalogs, each front-end subsystem also has a set of tables containing information specific to that subsystem. For example, VIFRED forms are stored in four different system catalogs. When the user references a particular form, as in the case of a QBFname query target, the front-end subsystem goes into these catalogs to retrieve the definition of that particular form. The *II_FORMS* catalog includes information on the overall form. It includes the object ID and the size of the form. It also has the number of fields and table fields on the form.

Each field in a form is defined in the *II_FIELDS* catalog. This table shows the sequence number and location of a particular field on the form. It also shows the data type, the length, and attributes for the field. Attributes include things like default values, visual attributes, and validation checks.

The *II_TRIM* catalog is used to keep track of trim information on a form. This is kept in a separate catalog from fields because trim does not have as many attributes, such as default values, as a field. This catalog contains the location of a particular piece of trim and the text.

The fourth catalog for VIFRED forms is the *II_ENCODED_FORMS* catalog. This catalog contains an encoded version of a form. The data are stored in a format that can be quickly retrieved and converted into an executable form in the FRS on a particular operating system that INGRES is installed on.

System catalogs are always used by the data manager and front-end subsystems to process queries and manage the user interface. General-purpose front ends can also be used to query the system catalogs. Two other uses of the system catalogs are for moving objects from one database to another and to develop custom applications.

To use a general-purpose front-end on the system catalogs, the user simply names the relevant system catalog as the query target. RBF could be used to develop a report, or QBF to browse the data interactively. For example, using QBF a user could obtain information about particular columns of a table, stored in the *II_COLUMNS* table. Figure 10-3 shows the information contained in the II_COLUMNS table. This illustration shows the definition of the emp_name column of the emp table in the database. The creation and last alteration date of the column are displayed, in addition to the column definition as a variable character data type with a length of 20.

All of the front-end systems have a copy utility that allows information to be transmitted from one database to another. For example, the *copyform* utility takes the definition of a form from one database and moves it over to another. Graphs, forms, applications, and even entire databases can be copied.

In the case of the copyform utility, an image of the form is constructed and the image can then be transferred over to the other database. In the case of entire databases, the *copydb* utility creates a series of SQL statements, known as a script. The script is then executed on the new target database to create new tables, permits, indices, and other structures and possibly to load in data from the old database.

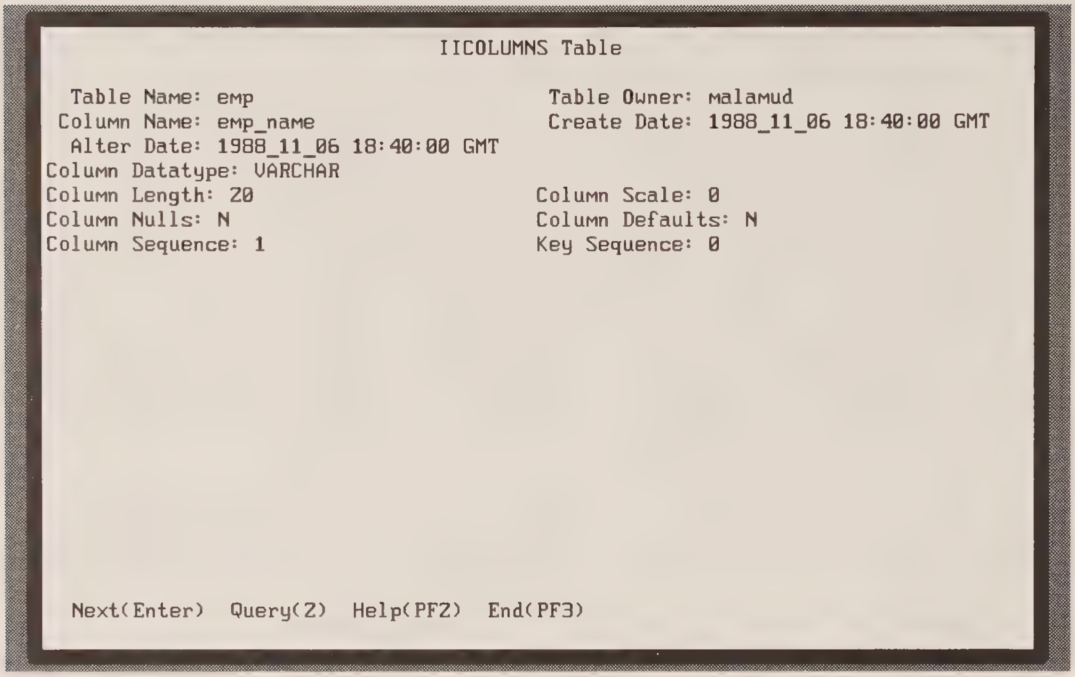
An option on all the copy utilities allows the file generated to be portable across different operating systems. This allows, for example, a form to be developed on a PC and then loaded into a large database system on a VAX. If the flag is omitted, the files are generated in an internal format for the operating system, which allows a more efficient transfer of the objects.

Users can also write their own applications that access the data dictionaries. For example, a developer may wish to present a user with a list of reports available from within an ABF application. To do this, the developer simply retrieves all relevant reports from the II_OBJECTS catalog into a table field. When the user selects an item, the user's INGRES 4GL code calls the report subsystem to display the report.

Extending the Data Dictionary

The INGRES system catalogs provide an extremely efficient method for managing the objects in an INGRES environment, including definitions of a few non-INGRES subsystems through the INGRES/Gateway capabilities. However, the system catalogs do not provide a fully general method for accessing data dictionary information in an environment with other kinds of subsystems, such as statistical analysis, project management, or manufacturing automation.

To solve this problem, the user can develop a series of INGRES tables that store meta-data about other types of subsystems. The problem with this approach is that each user is defining his own data dictionary. With no standard definition of meta-data and the ways to access that meta-data, it is hard to share information across different groups of users or applications.



```
                              IICOLUMNS Table

  Table Name: emp                          Table Owner: malamud
 Column Name: emp_name                     Create Date: 1988_11_06 18:40:00 GMT
  Alter Date: 1988_11_06 18:40:00 GMT
Column Datatype: VARCHAR
Column Length: 20                        Column Scale: 0
Column Nulls: N                          Column Defaults: N
Column Sequence: 1                       Key Sequence: 0
```

Courtesy of Relational Technology

Fig. 10-3 Using QBF to Examine the Systems Catalogs

The IRDS is a definition for a data dictionary that operates in a heterogeneous environment that applies to multiple DBMSs, as well as other subsystems, such as project management, documentation, or CASE tools. IRDS provides a definition for the tables that comprise a data dictionary and the services used to access those tables.

IRDS, because it is a standard, allows different systems to define data dictionary information. That information can then be moved from one implementation of the IRDS to another. DB2 data dictionary information, if stored in IRDS format, can be accessed by an INGRES application if it uses the IRDS services.

The key advantage of an IRDS-based environment is the extensibility of the data dictionary. If a user wishes to store definitions for a new type of information, say "projects," IRDS can be extended to include information about this new type of information. The definitions of tables and columns would now be stored in the same data dictionary as a project.

IRDS uses three concepts to define data:

- entities
- relationships
- attributes

Figure 10-4 shows a graphical illustration of entities, relationships, and attributes with objects from ABF. An entity, pictured as a rectangle, is the equivalent of the

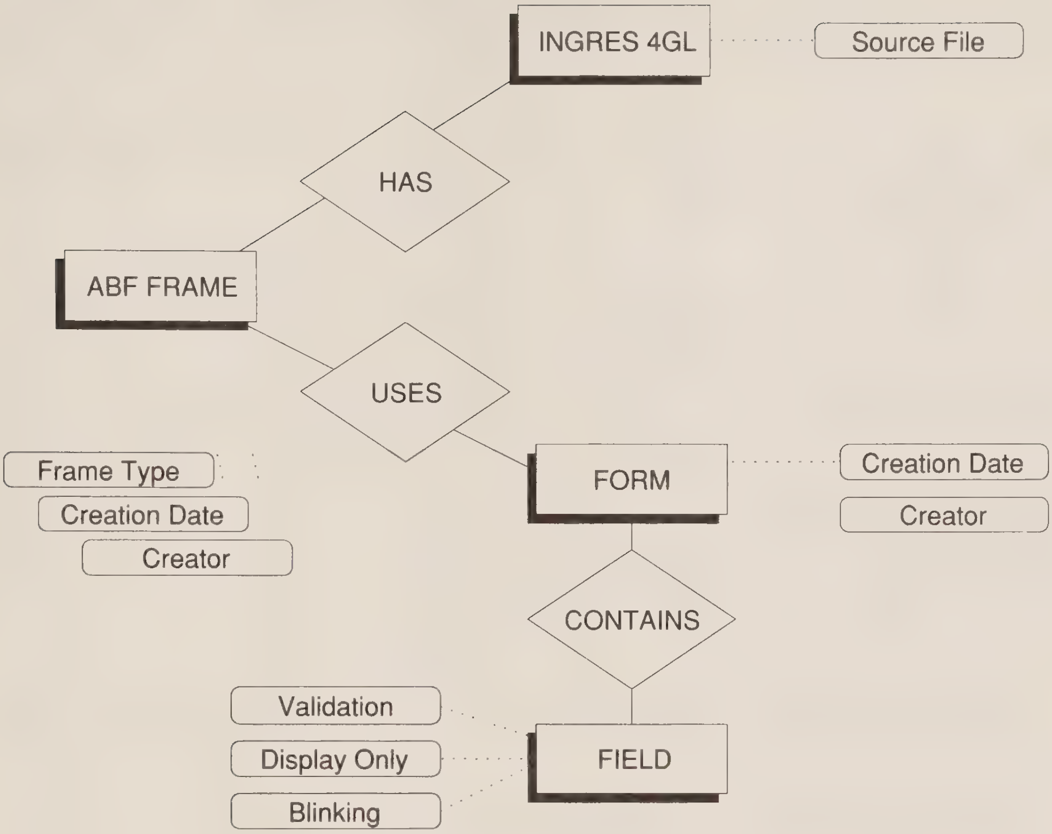


Fig. 10-4 Entities, Relationships, and Attributes

INGRES concept of an object. In ABF, frames, forms, fields, and INGRES 4GL code are all examples of different entities. Notice that INGRES 4GL code is not stored in the INGRES system catalogs, but is still considered an entity in IRDS. An entity could be the general concept of an INGRES table, or a particular instance of an INGRES table. An entity could also be a non-INGRES concept, such as a project or a document.

Entities have relationships with other entities. INGRES 4GL code, for example, is associated with a particular ABF frame. The frame also has a form. The form, in turn, has a set of fields. Both entities and relationships can have attributes. An ABF frame has a frame type, a creation date, and a creator. The INGRES 4GL code has a file that it is contained in. Figure 10-4 shows only a few of the attributes that are associated with the entities in ABF.

Attributes can be single- or multivalued. A multivalued attribute has several values for the entity or relationship it is associated with. A field in a VIFRED form can have the attribute validation check. Since only one validation check can be defined for each attribute, this would be a single-valued attribute. On the other hand, the field can have several display attributes, such as highlighting, underlining, or putting a box around the field.

The Dictionary Meta-Schema

The entity-relationship diagram in Figure 10-4 shows information about the relationship between the types of entities that make up an ABF application. It would also be possible to draw a similar diagram for a particular ABF application. Instead of the generic entity "Abf_Frame," the diagram would have the name of a particular frame, and the INGRES 4GL code and the name of a form associated with the frame.

The concept of entities and relationships can be used to represent data at various levels—the general types of entities that make up an ABF application and the specific instances associated with an actual ABF application, for example. The combination of the definition of the components of an ABF application and the instances of those components is known as a pair of data. The IRDS model has four levels of data, with any two adjacent levels of data making up a pair (see Fig. 10-5).

The definition of component types and the entities that fall within those types is the Information Resource Dictionary (IRD) pair of data; this is the information stored in the data dictionary. A lower-level pair is the name of an object, such as an ABF frame, and the actual object. This lower-level pair is known as the data pair. Another example of a data pair would be the data in an employee table of a user database. The actual data, together with the definition of the employee table in the system catalogs, are a data pair. The IRD provides a higher level of definition. The lowest level of the IRD is the employee table. The upper level of the IRD pair is the fact that a table has certain attributes, such as a creation date, and relationships with other entities, such as permits or integrities.

The IRDS model has yet one more pair of data. At the bottom of that pair are the types of entities in an IRD. The top level of that pair is known as the meta-schema, which defines how an IRD is structured. The meta-schema is what allows the IRD to be extended to introduce new types of entities, relationships, and attributes. Part of the meta-schema for the IRD, for example, would be defining that entities of type "INGRES TABLE" can have relationships with entities of type "PERMIT."

The fundamental IRD thus defines a set of concepts, such as entity types, relationship types, and attribute types. The IRD schema contains examples of entity types, such as INGRES TABLE or Abf_Frame. The IRD itself has instances of those entity types. Finally, the entities described in the IRD presumably exist someplace, as in the case of employee data being stored in a particular table in a particular database.

The basic IRDS model is thus an empty shell—a framework for defining meta-data. In addition to the IRDS schema, the basic ANSI standard is supplemented by a basic functional schema that implements some standard definitions for objects. For example, a file is an entity in the basic functional schema. Specific implementations, such as an

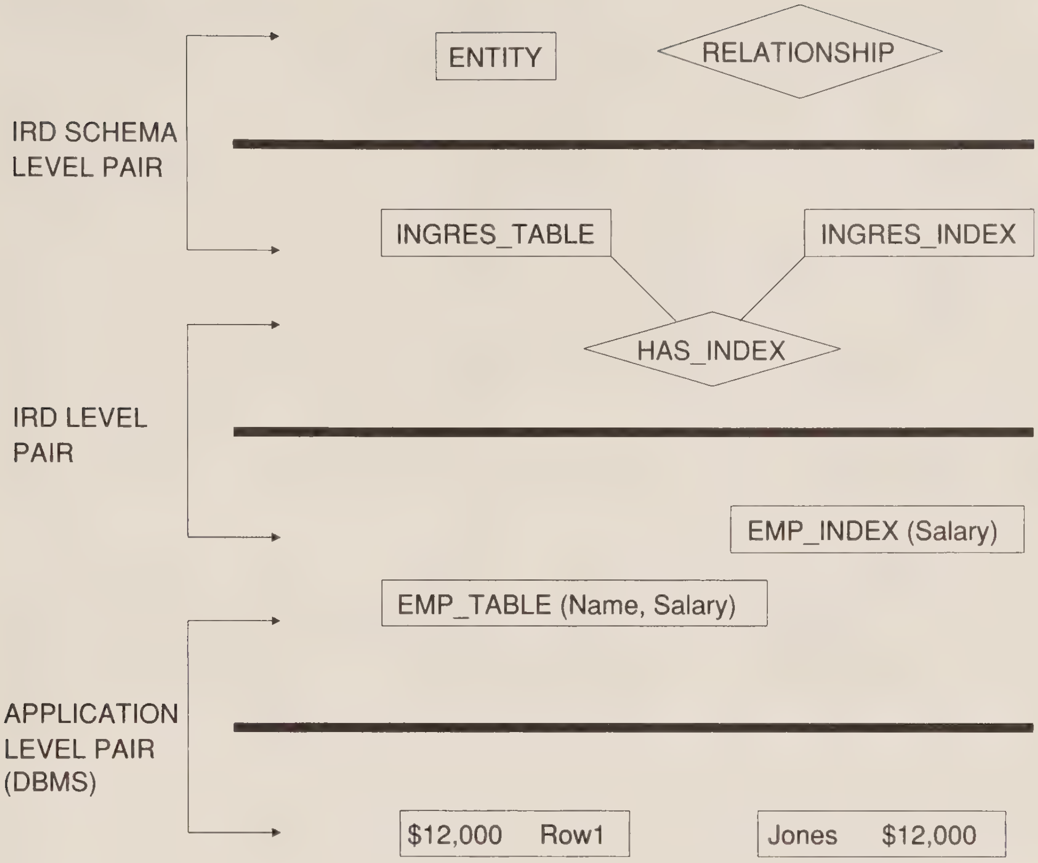


Fig. 10-5 Pairs of Data Definition in the IRD

INGRES-based data dictionary may also define some extensions. These implementor-defined extensions might include entities such as a database or a table. Finally, the user is able to extend the data dictionary to add other types of entities.

Storing Data in the IRD

In addition to the data model for the dictionary, the IRDS standard defines a set of rules on how information will be stored in the data dictionary. This controlled access to the data dictionary ensures the integrity of meta-data. The IRDS has three goals:

- extensibility of data content
- data integrity at all levels of the dictionary
- controlled access to data

IRDS offers a wide variety of different facilities. The basic facility is control over the naming of entities. There is a namespace in the IRD, and another, distinct namespace for the IRD definition. Every entity in a namespace has a unique name, known as an access name. In addition, the entity may have a longer name, known as the descriptive name. For example, a user could define a new entity type called Abf_Frame. The access name Abf_Frame must be unique. No other entity types should already have that name. The user could also define a descriptive name, such as "Application by Forms Frame Type." Users can then define a series of entities of type Abf_Frame. It is possible (though not recommended) to have an entity of type Abf_Frame have the access name Abf_Frame.

To provide some flexibility in the management of data, each entity can have different versions. The access name is the same for all different versions, but the entities also have a variation name and a revision number. For example, if we have a piece of data at the IRD level called EMPLOYEE, it is possible to have different variations of employee based on the different places it is located. One variation might be INGRES; another might be FORTRAN.

Naming rules can be defined for data in the dictionary. The naming rules are defined as a scan mask, much like the templates that were used for RBF output formatting. In a scan mask, the letter "A" matches any letter, the number "9" matches any number, and "*" matches any sequence of characters. To define a rule requiring that a name begin with a letter, followed by a number, followed by any character, the following scan mask would be constructed:

```
A9*
```

Any user entering a new entity of that type would be required to follow the naming rule before the entity could be stored in the data dictionary.

System-generated access names can be provided where users do not provide an explicit name (or a program generates several entity definitions automatically). A special attribute for each entity type defines whether the system will generate names automatically when users do not provide one. The system-generated names consist of a unique prefix for that entity type (an attribute for that entity type) plus a series of digits. No user-assigned access names can fall within the range of existing or potential range of system-generated access names.

Life Cycles and Views

Every entity in the IRD (and the IRD definition) is part of the life cycle control facility. The facility places each entity in a particular life cycle. The basic three life cycles are uncontrolled, controlled (production), and archived. There is an optional extended life cycle phase option that allows multiple partitions within the controlled and uncontrolled phases.

When a new type of data is being defined, it is placed in the uncontrolled life cycle phase. After the data become publicly available, they are moved into the controlled phase. Users searching for new data available would search the controlled partition of the data dictionary. After a type of data has outlived it's usefulness, it is moved to the archive phase.

Normally, security in the IRD is provided through a view. A view consists of a particular partition and permission to carry out different levels of operations on instances of specified types in that partition. Each user is then associated with a particular view of the dictionary. A view consists of a life cycle partition and a list of permissions saying what actions are permitted on what entity types. There is also a provision to prohibit users from seeing relationships in which one of the entities falls outside of the view. Users are allowed to delete, modify, add, and read entities as the basic permission classes. It is also possible to define other permission types such as the ability to secure or modify the phase of an entity.

The optional entity-level security facility allows a more granular method of controlling access. Each entity in the IRD (or IRD schema) can have one or several access controllers associated with it. The access controller has two locks, a read lock for all read-related operations and write lock for all operations. Each entity with an access controller is locked and may only be opened when the access controller presents the appropriate key.

Each view then has an access key given to it. By properly setting up a series of views and access controllers, the administrator is thus able to provide multiple levels of security. It is possible that a single access controller services multiple entities, and it is also possible that multiple access controllers be assigned to the same secured entity.

Quality indicators are used to supplement life cycle phases. Every entity has an optional attribute known as the quality indicator. The implementor or site administrator defines what are valid quality indicators. In a structured design environment, these quality indicators might include designed, coded, unit-tested, system-tested, and implemented.

The Services Interface

IRDS is based on a services interface, which consists of a set of functions that interact with the underlying data dictionary. Built on top of the services interface there may be several different types of user interfaces, including:

- command language
- panel interface
- export/import files
- user-developed applications

In the ISO version of the IRDS standard, all interfaces are required to use the services interface. Figure 10-6 illustrates the use of the services interface, which provides a bridge to the data repository that keeps the IRD information and ensures that operations on the data dictionary are properly carried out.

In the ANSI version, it is possible to implement the command or panel interfaces so that they directly access the contents of the data dictionary. Note that this approach requires each of the different interfaces to implement the same functionality. Most users will thus develop a variety of functions (the services interface) and use those functions in the panel, command language, or any other interfaces.

A command language interface is an application that allows the user to use an IRDS-specific command language instead of basic SQL to access and modify the data dictionary. A version of the command language interface is the application programming interface, which allows the programmer to embed command language statements in program code, just as SQL is embedded in program code. The panel interface consists of a menu-driven program that is used to access the data dictionary. This interface, much like QBF, is used to browse and change dictionary data or meta-data. A panel interface can be fairly easily constructed using INGRES 4GL and ABF.

The IRD-IRD interface is used to import and export data dictionary information to another dictionary. A standard format is defined for the interchange of information. The import/export interface generates a file that is moved over to the system with the target data dictionary. The data are then moved into an uncontrolled life cycle partition. Users can then move these data definitions over to other life cycle phases.

All of these applications use the services interface to access the data dictionary. The services interface is also available to other types of applications. For example, a CASE environment could use the services interface to store model elements for the designs of information systems. To use the services interface, the user starts a session with the IRDS. A series of transactions is then carried out. A transaction is not made permanent until the user commits those steps. It is also possible to abort an ongoing transaction by issuing a rollback call. Finally, the user closes the IRDS.

Notice that the interaction with the IRDS is very similar to interaction with an SQL-based database. The services interface consists of a standard series of operations such as adding a new type of entity. These services are equivalent to a series of SQL statements that operate on a DBMS, the repository for the IRD. This does not mean that the underlying implementation of the IRDS is in fact a DBMS. However, in most circumstances, one can expect the IRDS implementer to use the services of an SQL-based data server. It is possible, using the ISO Remote Data Access (RDA) standards or INGRES/STAR, that the data server is in fact several different data managers distributed in multiple locations.

Before a user can start an IRDS session, he must be predefined to the system. This consists of creating an instance of the entity type *IRDS-USER*. This entity then has a series of relationships to other entities of types such as *IRD-VIEW* or *SCHEMA-VIEW*. The particular view, in turn, points to a particular partition of the system.

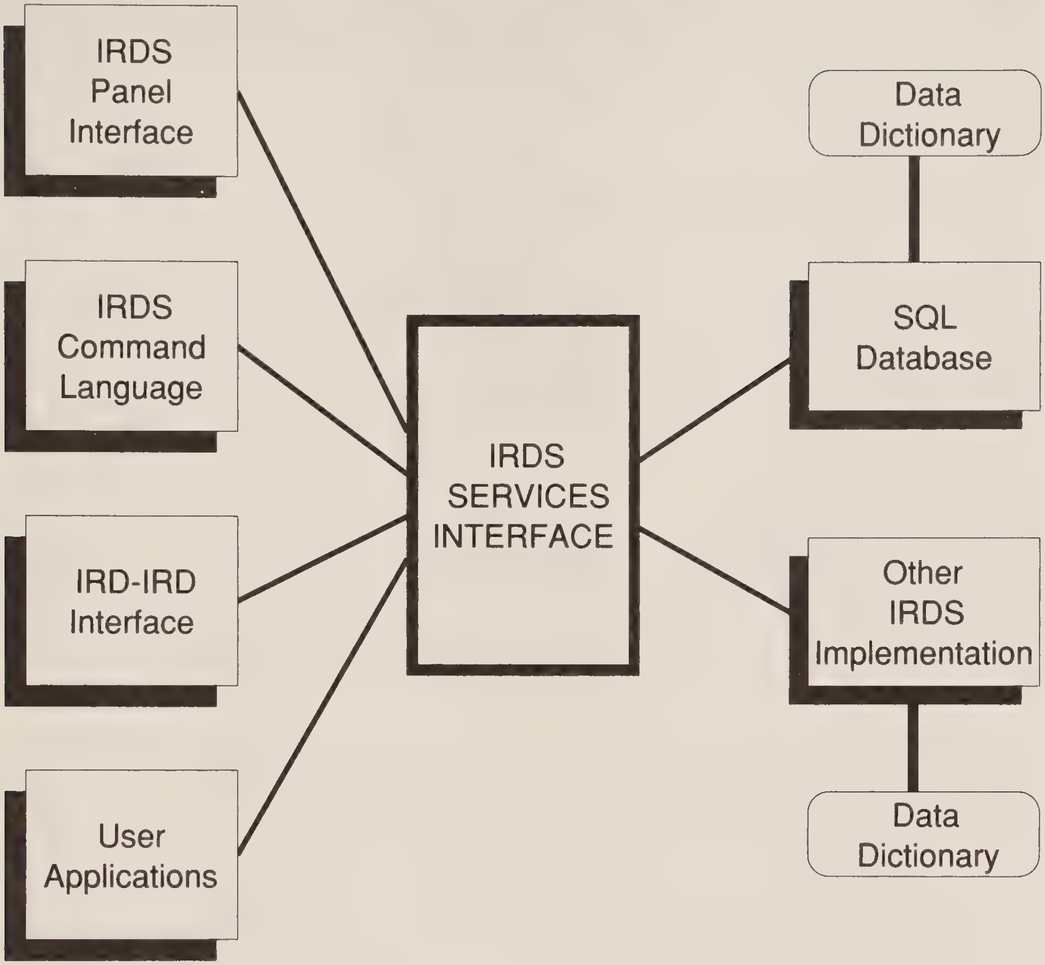


Fig. 10-6 Access Methods to the IRDS

IRDS services consist of navigating the entity-relationship structure of the dictionary. The user first finds a particular entity, relationship, or attribute, known as establishing a position. Once a position is established, the user can read or write the object. The user can also define a new search, such as finding all entities that have a particular type of relationship with the entity at the current position. The new search yields a set of entities, and the user can establish a new position on any of those. After the current search is exhausted, the user is returned to the previous position on the first entity.

Establishing a position uses a cursor mechanism similar to SQL. The user defines a set of entities or relationships that she is interested in based on search criteria. The criteria can be based on the key value, or a more complicated set of search criteria. For an entity, a search on key value might consist of all versions of a particular access name, where the access name consists of a scan mask. Or, the user can request a particular version consisting of the highest revision of a particular variation name.

Uses of a Data Dictionary

A data dictionary allows information to be defined in a consistent manner and the definitions to be easily accessed. One of the problems in a large computing environment is that there are many pools of knowledge, each implemented in a different database or portion of a single database. Often, information is not stored in a database, but is kept in a filing cabinet or on a PC.

The data dictionary allows users to find what information exists. Note that data dictionary security provisions also allow the data administrator to prevent users from finding out what information exists. There is a big difference, however, from making a conscious decision to restrict access information and from having restricted access to information because people do not know where to go to find it. The data dictionary is thus a fundamental tool for end users that are analyzing information. The dictionary tells them where to go to find data and what tools are available to access that data.

Data dictionaries are also vital for defining new types of information. A programmer can check the data dictionary to see if particular data already exist, and how they are defined. The programmer might then store that information in another location, and the definition in the data dictionary as a variation on existing data.

The data dictionary is also a useful tool for documenting existing applications and databases. Since the definition of the database tables are in the data dictionary, it is fairly simple to define a report or ABF application that accesses this information and presents it to the users. The INGRES/Menu tables catalog is an example of an application that uses the services of a data dictionary to present information to the end user.

A related documentation example is an impact of change report. If a user wishes to change the definition of a piece of information, the data dictionary can be queried to find all applications and data repositories that use that particular piece of information. An impact of change report allows application developers to quickly determine the effect of changes in the data environment on users and applications.

Another documentation example would be to use the data dictionary to present the structure of an INGRES application. Applications consist of frames and these frames are of various types and usually include a form. The user could construct a report showing all forms in the application and which applications use them.

One problem with data dictionaries is that most organizations have a variety of existing applications that have not been defined in the dictionary. Programs can be written that scan the code of existing applications and enter the relevant information into the data dictionary. Such a program is called a backloader because it takes an existing application and loads a definition into the data dictionary.

In the case of a FORTRAN program, a backloader would scan the source code and look for all subprocedures, functions, variable definitions, and other components of the program. This information would be loaded into the data dictionary. Then, a report can be constructed that shows the different elements in the program, how they are defined, what parameters they use, and other structure information. Another backloader might be used to document the way a particular system is configured. On VMS, for example, there are a variety of files and system catalogs that can tell the user what disk drives are present, how much memory is available, the version of the operating system, and currently validated user. This information could be automatically loaded into a data dictionary and used by the network administrator.

A more futuristic use of the data dictionary is to actively use it in the administration of computers on a network. In addition to defining the configuration of various computers, the network administrator would also define the commands used to administer a computer. An example is the command that is used to provide checkpoints of INGRES databases. This is a command that is run periodically. The checkpoint command has a series of parameters, which would be defined in the data dictionary as parameters of the command. One of the attributes would be the periodicity of the checkpoint command—how often it is run. Next, relationships can be set up between the checkpoint command and the targets of that command—particular databases. Other commands, such as adding new users or sending out bills on system utilization from the accounting files, can also be defined in the data dictionary.

Once commands for administration are defined in the data dictionary, it is a fairly straightforward proposition to write an application that goes and looks for tasks that are ready to be run. For each of the commands, the application looks at the periodicity, the parameter list and targets, and constructs a command and runs it. Such a program could provide the administrator with a flexible tool for defining new administration functions. The functions could then be run automatically (using the periodicity factor), or could be manually activated using a menu-driven application. The key is that a new application does not have to be written just because a new command is defined—the same general-purpose application is used to activate all functions.

This use of a data dictionary is not quite as far-fetched as it may sound at first. DEC uses a similar concept, called the Enterprise Management Architecture (EMA), for the management of different components on a network (see Fig. 10-7). The key to EMA is a data repository that includes the definitions of three types of components:

- protocol modules
- functional modules
- presentation modules

A protocol definition is the interface to a particular type of environment. DECnet Phase IV uses one set of protocols; DEC terminal servers use another protocol. Each of these protocols is defined to the data repository. The user can likewise define a new set of protocols, such as the INGRES database administrator protocol.

Independently defined from the protocol definitions are function definitions. “Turn_On” is an example of a function. This function can then be applied to a variety

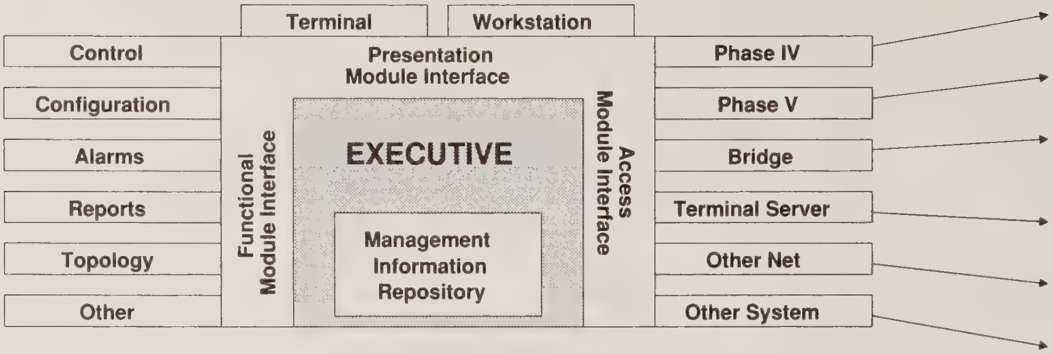


Fig. 10-7 DEC's Enterprise Management Architecture

of different entities, each one corresponding to different protocols. The Turn_On command could reboot a VAX, initialize DECnet on another node, and start the INGRES server, all at the same time.

The third set of functions in EMA are presentation modules. A different presentation module is defined for different types of devices, such as VT100 terminals, bit-mapped VAXstations, or even an IBM 3270 terminal. Neither the protocol definitions nor the functional modules are tied into the presentation modules.

Summary

Data dictionaries are a vital tool for managing an environment consisting of a large number of applications and data repositories. The data dictionary allows new forms of information and applications that access and display that information to be defined and controlled.

The INGRES system catalogs are an example of a data dictionary. Whenever a new object, such as a table or form, is defined, the information is entered in the INGRES system catalogs. Using the services of the database to manage the system catalogs provides control over the definition of the objects. Because the meta-data are stored as a series of tables, general-purpose tools can access the information using SQL statements. Since meta-data are stored as tables, it is easy to move the definitions from one database to another.

Using tables to store meta-data also allows the data dictionary to use all the services of the data manager. New objects cannot be created that conflict with existing ones, as in the case of two tables with the same name. Efficient access to meta-data is available through the query optimizer. Locking ensures that two users do not simultaneously change the definition of one object.

More general data dictionaries, such as IRDS, can be used by a variety of other users. All of these data dictionaries allow coordinated, structured access to the definition of data elements. These data elements can be the traditional data elements—tables in a database and their columns. They can also include much more general concepts such as front-end objects, tasks in a project, or the definition of commands that administer a system.

With data dictionaries implemented under INGRES, the services of INGRES/STAR can be used to present a single logical view of several locally administered data dictionaries. This allows a distributed data dictionary to be used in conjunction with distributed data. This chapter has thus examined two types of data dictionaries. The fairly specific INGRES system catalogs are optimized for storing INGRES-related objects. This data dictionary has the attribute of being very efficient for INGRES-related work.

The IRDS is a much more general model of a data dictionary. The model needs to be able to accommodate FORTRAN- and COBOL-related concepts as well as INGRES and other database vendor's objects. Because of its generality, IRDS provides the ability to incorporate a wide variety of elements into a single dictionary.

Both forms of data dictionaries are likely to be used in most environments. IRDS, as a fairly new standard, will take a while to stabilize and gain acceptance in the user and vendor communities. As IRDS matures, it will offer the promise of a standard way to access meta-data from a wide variety of programming environments and toolsets.

Chapter

11

Database Design and CASE Tools

INGRES/team*work* CASE Environment

INGRES/team*work* is a set of tools that allows users to analyze and design information systems. These tools, in contrast to many computer-aided software engineering (CASE) environments, are multiuser in nature, allowing many different programmers and analysts to work on a complex set of specifications while insuring that different users actions do not conflict with each other.

Relational Technology and CADRE have signed a joint development agreement that brings the team*work* set of tools directly to bear on the questions of database development. This joint development agreement allows current team*work* components to be used for the design of INGRES-based applications and also extends the team*work* tools to integrate them with the applications development and physical database design portions of INGRES.

Team*work* consists of several different modeling tools. These tools allow a broad picture of the organization to be developed as an information model, similar to the entity-relationship diagram examined in the previous chapter. Next, the components of this model can be moved into a systems analysis modeling tool for further refinement. Finally, the analysis model can be handed off to a design team that defines the different modules in an information system and how they function. All of the components of the different modeling tools, such as entities and relationships in the information modeling environment, are stored in a data dictionary. The elements defined in one model are available to other users and other modeling tools.

All of the tools in team*work* are graphical in nature, allowing the designer to visually display the relationship between different data dictionary elements. The graphical display of information is quite important for the efficient and complete design of an information system. A strictly textual description quickly degenerates into very long, unreadable documents.

Graphical displays allow the information to be structured. An example is the data flow diagram in which information flows between different processes are displayed. Each of the processes can in turn be a data flow diagram that shows a more detailed flow of data within that process. Hiding the underlying details of the process allows the designer to concentrate on the overall structure of the system and then look at the underlying details.

An important attribute of team*work* is that it is an extensible environment. New tools can be easily added that access the data dictionary. This is important for two reasons. First, there are a large variety of different design methodologies. While team*work* supports three of the most important, there are always new ways of doing structured design and development. New graphics editors can be designed that reflect these various design philosophies.

The second reason for extensibility is that many functions not traditionally provided in a CASE environment are needed if the logical designs are to be quickly translated into a working information system. A structured design model can specify the logical design of the data in a system. To implement this logical store as an INGRES database requires translating the logical information into the underlying tables, views, storage structures, and other components of the database.

The extensibility of team*work* allows Relational Technology and CADRE to provide this next level of support for a CASE environment. The join development between the two companies focuses on how a logical information system design can be quickly translated into a working database and prototype applications.

Information Modeling

Team*work*/IM allows the user to develop an entity-relationship diagram for an information system. This step is often followed by use of two other INGRES/team*work* tools for systems analysis and systems design. The team*work* information modeling tool (sometimes known as a Chen Entity-Relationship Model) is used to generate an overall view of the information system. The entity relationship diagram represents an information system solely in terms of the data within the system. The systems analysis tool allows this information to be supplemented with descriptions of the processes that affect that data. Although a variety of different methodologies are available, the information model is usually the first step in the design process because it represents a very high level view of the information in a database.

An information model consists of entities, relationships, and attributes. Each of these objects in the model is stored in a data dictionary. The data dictionary entries are then available for use in the other modeling tools within the team*work* environment.

An entity is an object in the real world that is being modeled in the information system. Employees or managers are two examples of entities. Entities are then connected to each other with relationships. An employee might have the relationship "works for" to the entity manager. Note that at this point we are not concerned with how these entities are stored in a database. The fact that an employee is also a manager could be stored in the same table as the employee information in a relational database. At this point, we are more concerned with the logical design of the information system. At a later point, this logical design will be translated into a physical design that corresponds to tables in a database or records in a file.

Each relationship in the model has a name and a cardinality. The name is simply a text string attached to the object. Cardinality denotes how many instances of an entity participate in a relationship. For example, the employee portion of the works-for relationship might have a cardinality of 1, denoting that every employee works for one manager and only one manager. Alternatively, the cardinality of the relationship could have a value of 0/1, denoting that an employee has at most one manager, but could have no managers. A 1/n relationship denotes that the employee has at least one manager, but could have several different managers.

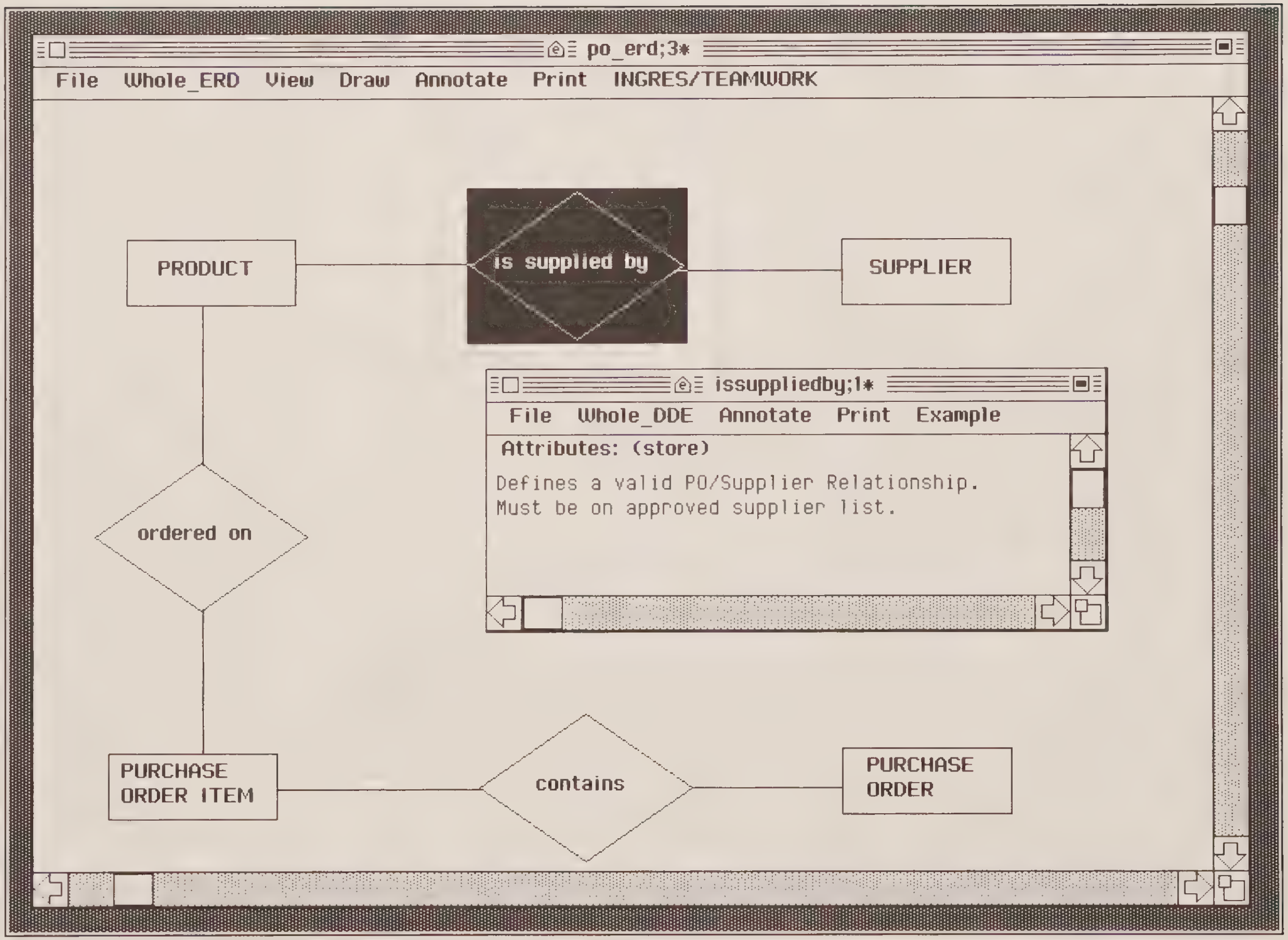
The other side of the same relationship can have a different cardinality. Even if the employee has a cardinality of 1, the manager could have a cardinality of 0/n. This means that a manager could have no employees assigned, or many employees assigned. The translation of the two sides of the relationship into business terms is as follows: Every employee has exactly one manager and a manager can have any number of employees (including none).

A dependent entity is one that only exists when a certain relationship exists between two other entities in the model. For example, a purchase order can only be created when there exists a purchaser and a vendor on the approved vendor list. If either of those two pieces is missing, the entity purchase order does not exist (at least in the world of the information model).

Both entities and relationships can have a series of attributes assigned to them. The entity employee could have the attributes last name, first name, telephone number, and salary. Every entity needs at least one attribute that uniquely identifies it—equivalent to a primary key in the relational model.

Figure 11-1 shows an example of an entity-relationship diagram constructed with team*work*/IM for a purchase order database. The model shows suppliers, products, and purchase orders. Purchase orders are broken down into two entities: purchase order items and the purchase order itself. This model was constructed using the graphical editor of team*work*, which allows the user to construct a new entity-relationship diagram or edit an existing one. For example, on an existing diagram, the user can add a new entity and form a relationship to existing parts of the model.

To add attributes to either entities or relationship, the user simply points to the desired object using a mouse. Clicking the mouse button selects that object and a menu is displayed. At this point the user could move the object, delete it, or add attributes. To add an attribute, the user clicks on the appropriate menu option and a small window opens up on the screen. At this point, the user enters textual information that describes the object and any integrity rules associated with the object. In Figure 11-1, an integrity



Attributes: (store)

Defines a valid PO/Supplier Relationship.
Must be on approved supplier list.

Courtesy of CADRE

Fig. 11-1 Entity Relationship Diagram

rule is that a product can be ordered from a supplier only if the supplier is on the approved supplier list.

Note that there are a variety of different ways that this model could be implemented. The integrity rule, for example, could be implemented in the front-end applications or as a back-end integrity constraint. The entities and relationships can be combined into a variety of different database tables. All of these issues are dealt with at a later stage in the design process when a logical design is translated into a physical design for the database and the applications.

As in all the team*work* editors, the user has a variety of options that control how information is stored and displayed. For example, the user can zoom in or out on the diagram to see different levels of detail. The user can print the entire diagram or the portion currently displayed on the screen and can also undo any operation previously performed.

As in the other team*work* components, the entity-relationship diagram editor includes a consistency checking feature. Checking the syntax of a diagram assures that all objects have a name and that each object is properly constructed. For example, each entity must participate in at least one relationship and each relationship must be attached to at least two entities.

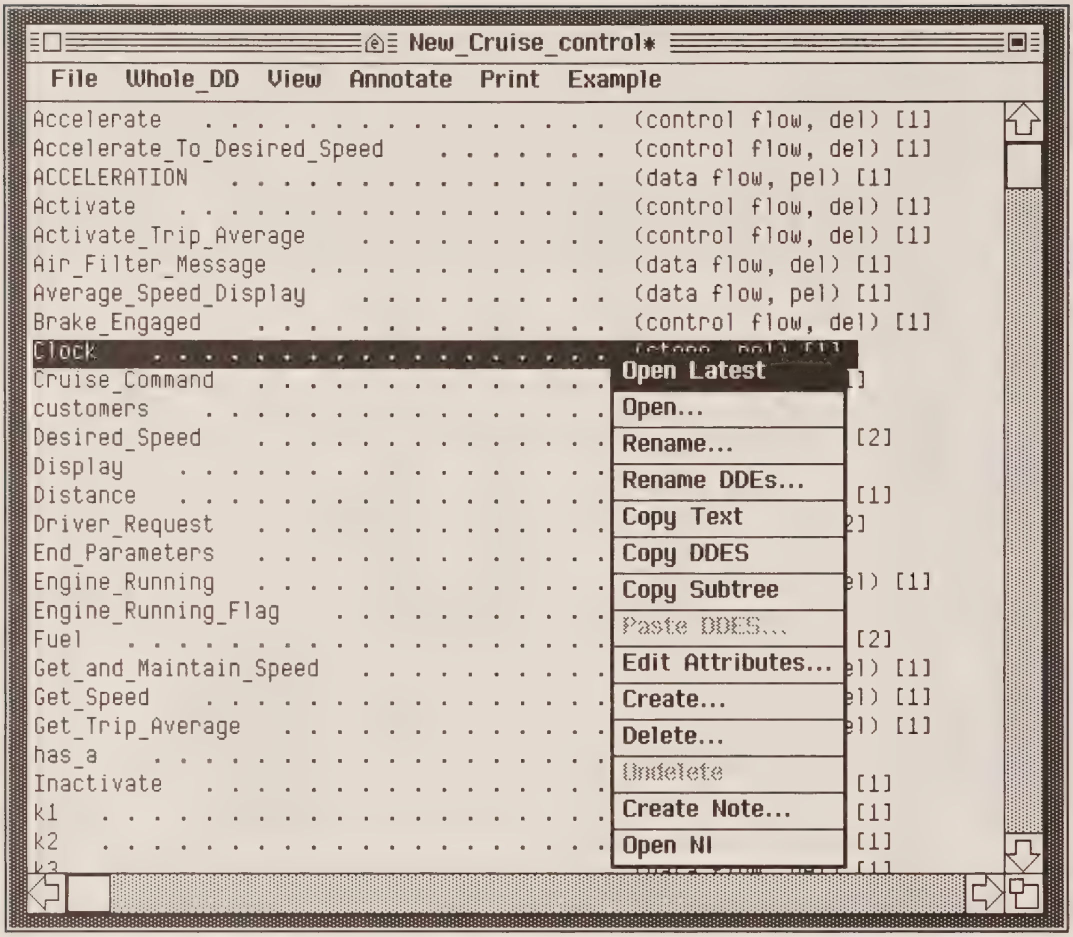
A further level of checking ensures that every object in the entity relationship diagram is also properly defined in the data dictionary. The most thorough level of consistency checking makes sure that the entities and relationships are fully normalized. Normalization, discussed in more detail later in this chapter in the context of normalizing database records, ensures among other things that objects do not contain repeating groups or circular definitions. The results of a syntax check can then be displayed on the screen or spooled to a printer.

The information model is used to construct a very broad picture of an information system. No attention is given at this step to the way that data are processed. For example, the purchase order entity-relationship diagram has no provision in it for how a purchase order is processed.

Systems Analysis

The team*work*/Systems Analysis (SA) is the next step in the design process. Systems analysis takes the entity-relationship model of the information system and restructures it in terms of flows of data and the processes in the information system that work with that data. Such a model is called a data flow diagram.

The systems analysis phase moves the focus from the data that an information system has to how those data are processed. All of the entities and relationships in the information model become data stores for the systems analysis phase, which shows how data moves between those different stores. Systems analysis thus concentrates on applications. A data flow diagram is a pictorial representation of an application that works with data stores. The application represented by the data flow diagram can in turn be



```
Accelerate . . . . . . . . . . . . . . . . . (control flow, del) [1]
Accelerate_To_Desired_Speed  . . . . . . . . (control flow, del) [1]
ACCELERATION . . . . . . . . . . . . . . . . (data flow, pel) [1]
Activate . . . . . . . . . . . . . . . . . . (control flow, del) [1]
Activate_Trip_Average  . . . . . . . . . . . (control flow, del) [1]
Air_Filter_Message . . . . . . . . . . . . . (data flow, del) [1]
Average_Speed_Display  . . . . . . . . . . . (data flow, pel) [1]
Brake_Engaged  . . . . . . . . . . . . . . . (control flow, del) [1]
Clock  . . . . . . . . . . . . . . . . . . . (store, pel) [1]
Cruise_Command . . . . . . . . . . . . . . .                    ]
customers  . . . . . . . . . . . . . . . . .
Desired_Speed  . . . . . . . . . . . . . . .                    [2]
Display  . . . . . . . . . . . . . . . . . .
Distance . . . . . . . . . . . . . . . . . .                    [1]
Driver_Request . . . . . . . . . . . . . . .                    2]
End_Parameters . . . . . . . . . . . . . . .
Engine_Running . . . . . . . . . . . . . . .                    el) [1]
Engine_Running_Flag  . . . . . . . . . . . .
Fuel . . . . . . . . . . . . . . . . . . . .                    [2]
Get_and_Maintain_Speed . . . . . . . . . . .                    el) [1]
Get_Speed  . . . . . . . . . . . . . . . . .                    el) [1]
Get_Trip_Average . . . . . . . . . . . . . .                    el) [1]
has_a  . . . . . . . . . . . . . . . . . . .
Inactivate . . . . . . . . . . . . . . . . .                    [1]
k1 . . . . . . . . . . . . . . . . . . . . .                    [1]
k2 . . . . . . . . . . . . . . . . . . . . .                    [1]
k3                                                              [1]
```

Courtesy of CADRE

Fig. 11-2 Index of Processes in the Data Dictionary

made up of other applications. All of these applications are known as processes and are stored in a process dictionary in team*work*.

To edit a data flow diagram, the user opens up a process index window on the workstation display (see Fig 11-2). To edit a particular process, the user points to it with the mouse and selects the object by clicking the mouse button. At this point, a window opens that gives the user the option to open the model and display its contents. Notice that the process index menu has two options to open a model. By default, the latest version of a model is opened. Team*work* also stores several versions of a component, and it is possible to open a previous version.

The user can also add a note that describes the process. Throughout the team*work* environment, any component can have a note attached to it. The notes can then be used to generate documentation on all the components of an information systems design.

Figure 11-3 shows an example of a data flow diagram that models a cruise control system for an automobile. The diagram shows three types of components:

- terminators
- data flows
- a process

A terminator is an object outside the scope of the current analysis. In Figure 11-3, gas stations, brakes, drive shafts, and other physical components of the automobile are not part of the cruise control system. They do, however, interact with the cruise control system by sending and receiving data.

The data flows are represented by arrows. For example, the fact that the brake is engaged is an example of a data flow. The cruise control process needs to monitor this indicator in order to deactivate the cruise control mechanism. The last object, the circle in the middle of the diagram, is a process. The process receives and sends flows of data to the terminators. There are two kinds of processes in systems analysis. A data flow diagram is itself a process, and data flow diagrams can be nested many layers deep. A process specification is the lowest level and contains a textual description of the steps involved in a particular function.

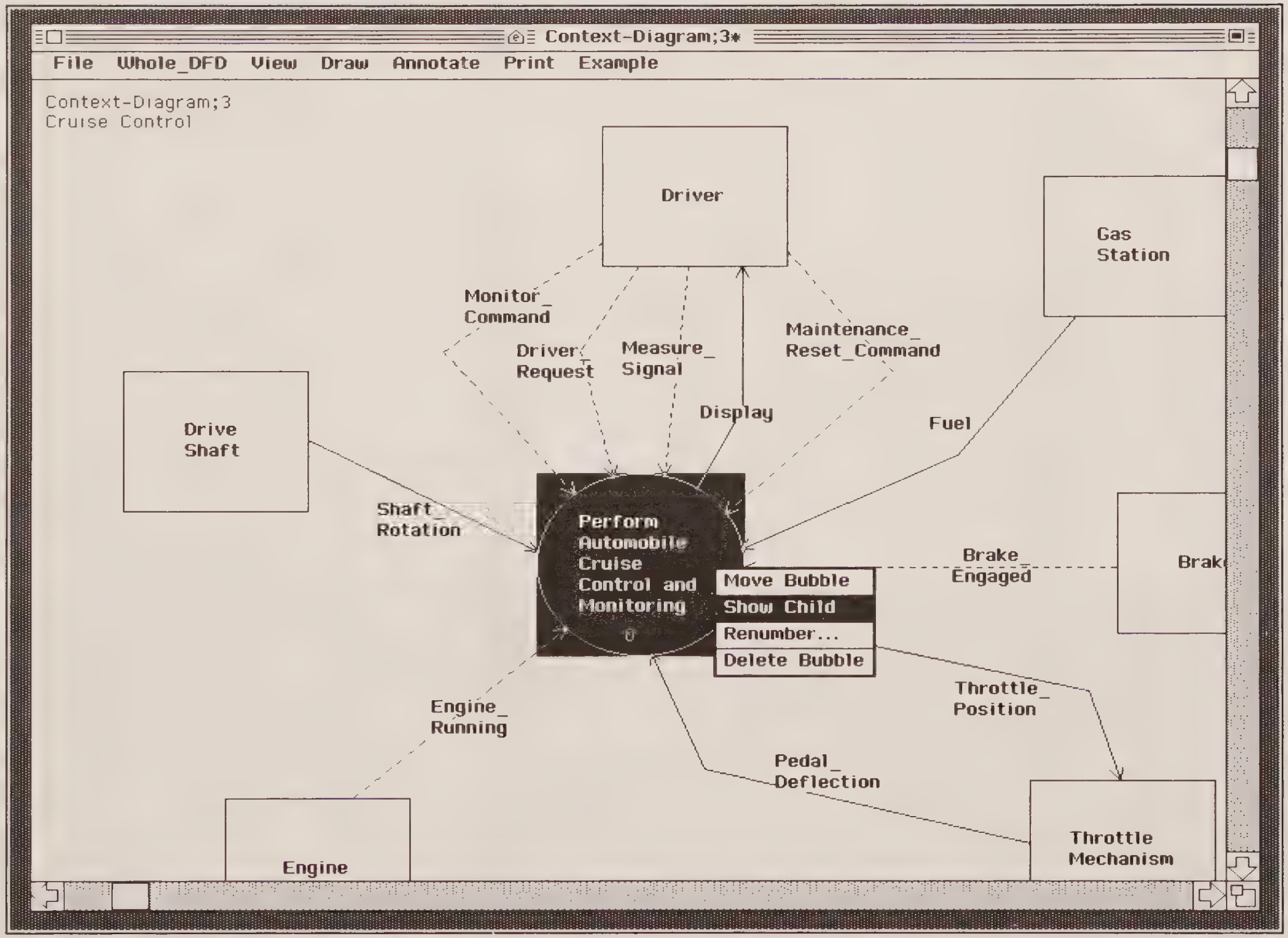
The diagram in Figure 11-3 is known as the context level of a data flow diagram. The context level is the only level that can contain terminators, which are outside of the scope of the current analysis. Figure 11-4 shows the next level of the data flow diagram, containing further details on the cruise control process, which contains several subprocesses. Notice, for example, the process to determine speed, which gets data flows from two sources. The shaft rotation data flow comes from the drive shaft terminator in the previous level. The other data flow comes from the pulse-count data store, represented as two lines on the diagram.

Data stores are equivalent to entities and relationships from the information modeling tool. By clicking on the data store, the user can open a window that shows the entity-relationship diagram associated with that store. In addition, the data store has associated with it a data dictionary entry (see Fig. 11-5), which contains the attributes of the store and a textual description. Notice that the data dictionary entry is the same as used in the entity-relationship diagram.

The reusability of model elements is an important part of the team*work* environment. Users can start by building a high-level model of the information used in an organization. Next, the systems analysis tool can use those model elements to provide more detailed specifications of how processes use the data stores to generate more information.

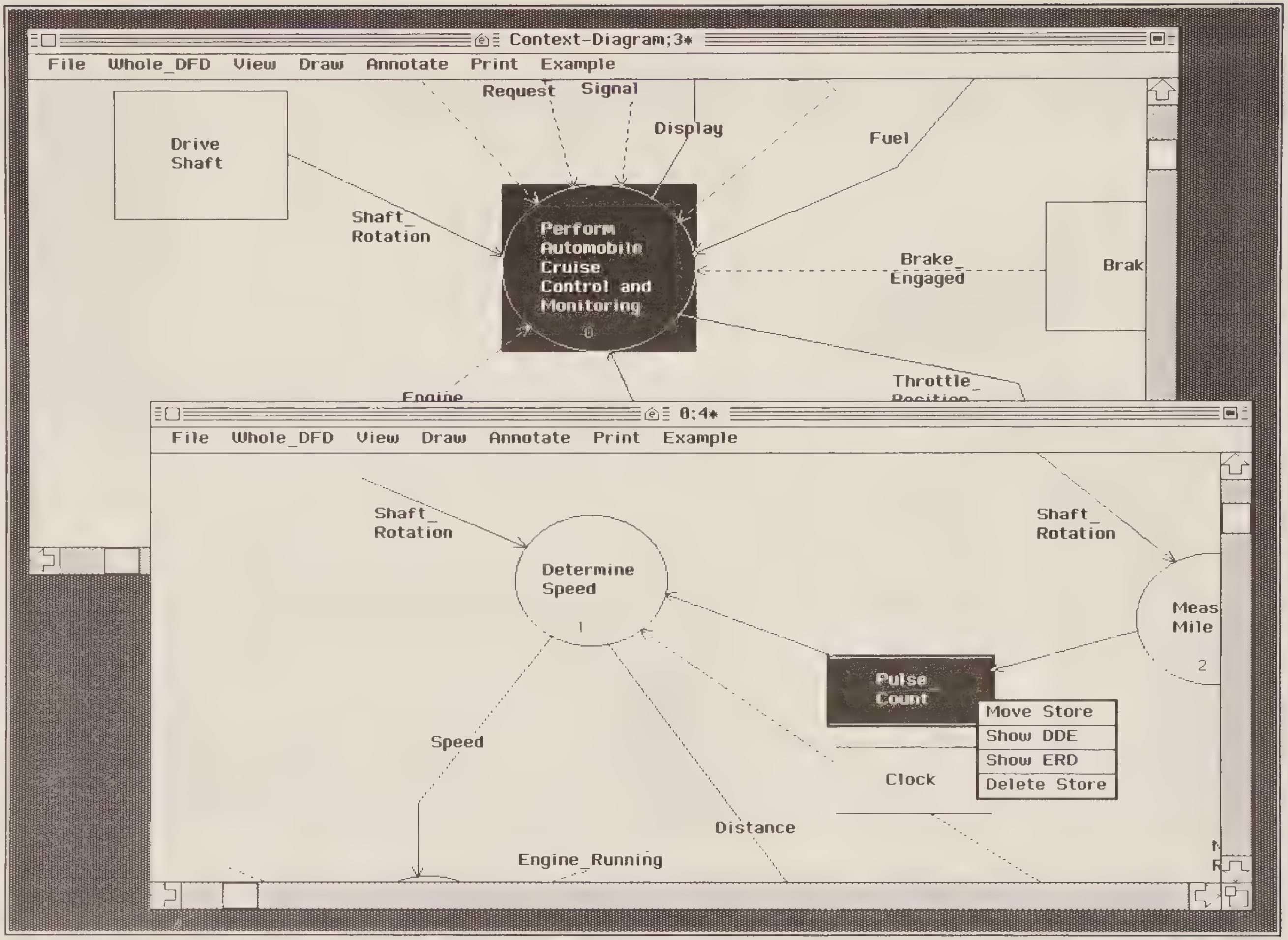
Once the data flow diagram is constructed, the user can check to make sure that the syntax is correct. This process consists of making sure that all the portions are correctly formulated. For example, the syntax checker will ensure that all data flows are connected on at least one end to a process. That process can represent a process specification or a lower-level data flow diagram.

Syntax checking also ensures that all processes have at least one data flow in or out. This makes sense as a process without any data flow really does not belong in the information system. The syntax checker can also make sure that the data flow diagram is balanced, checking that all the data flows that enter a child data flow diagram are



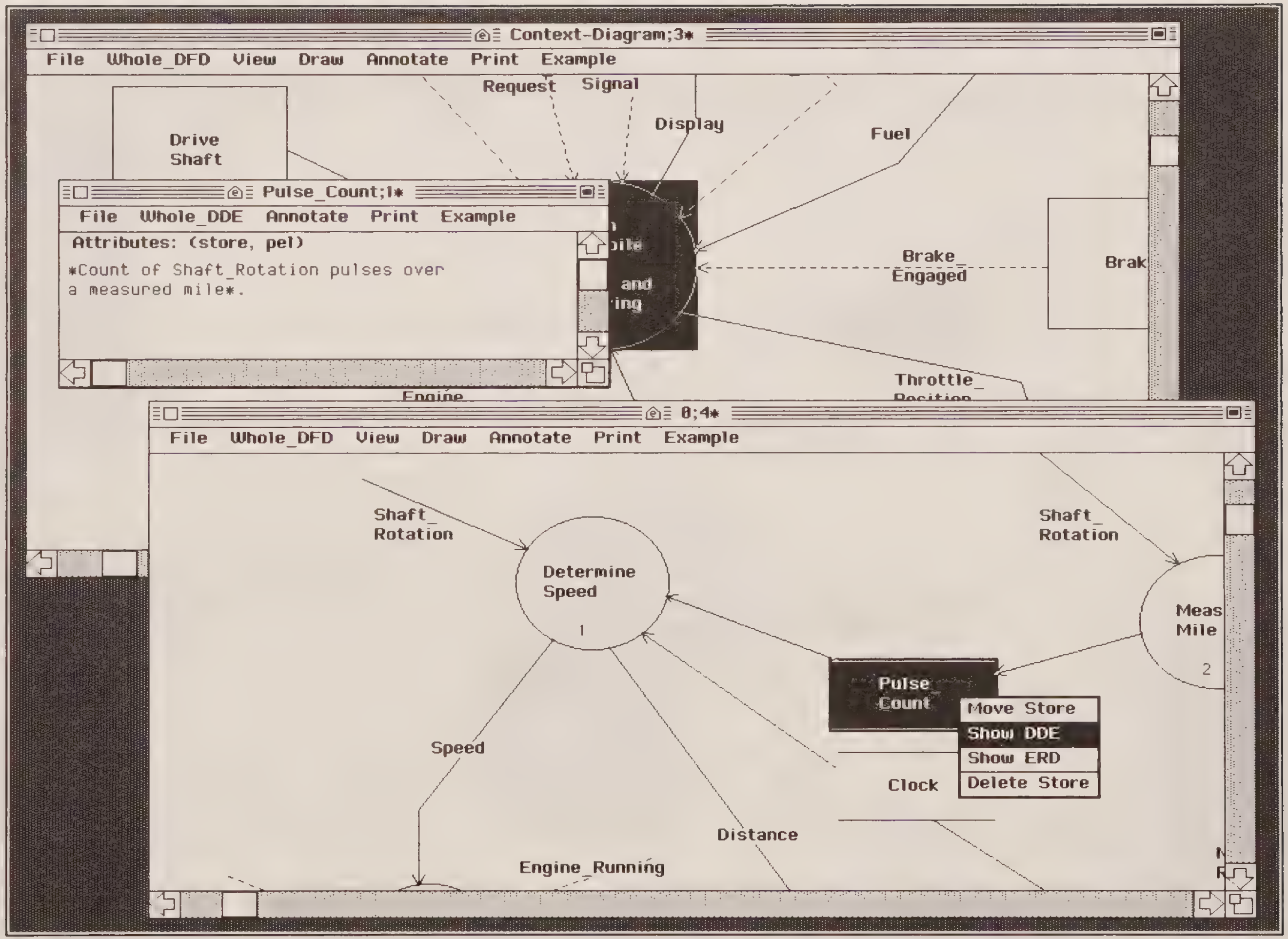
Courtesy of CADRE

Fig. 11-3 Data Flow Diagram Context Level



Courtesy of CADRE

Fig. 11-4 Lower-Level Data Flow Diagram



Attributes: (store, pel)

Count of Shaft_Rotation pulses over a measured mile.

Courtesy of CADRE

Fig. 11-5 Data Dictionary Entry for a Data Store

shown on the parent process that represents that data flow diagram. If this condition is violated, then data enter the process and are never used.

The report produced by the syntax checker can be examined visually in a separate window or spooled to a printer. The user has a variety of options available, such as checking all levels of the data flow diagram or only a portion at one level.

An extension to the team*work*/SA tool is team*work*/RT (real-time). Team*work*/RT adds two more concepts to the data flow diagram. Control flows are a form of data that represent actions or events. These events or actions control the activation of a particular process. Thus, in a factory environment, a control flow can ensure that a particular piece of machinery is activated before the process is activated that starts moving data into that piece of machinery.

A control specification details how various types of control information are processed to reach a particular decision. The control specification is a matrix of all combinations of input control flows, and the actions to take for each combination.

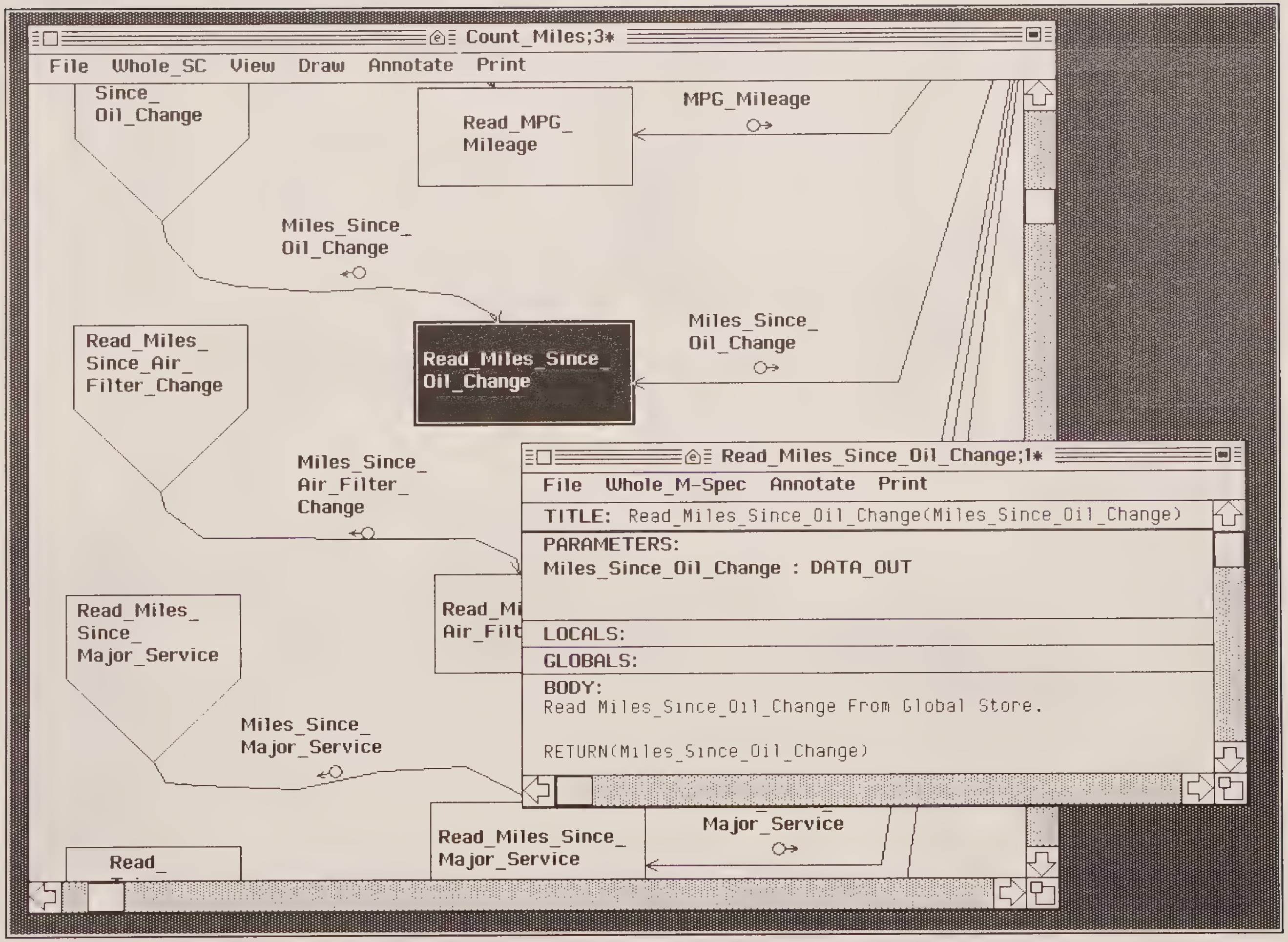
Systems Design

After the systems analysis phase, the project is moved over to the systems design process, which produces detailed specifications for all of the pieces of the application. The team*work*/Systems Design (SD) tools allow the designer to produce detailed specifications for each of the modules in a program. The model for systems design is known as a structure chart (see Fig. 11-6). The structure chart consists of modules and data stores. Each module has associated with it a module specification.

In Figure 11-6, the module specification displayed is for the module that reads the number of miles since the last oil change. The module specification has a title, parameters, various local and global variables, and a body. The body consists of pseudo-code. Pseudo-code is a way of representing steps in a program in a high-level, language-independent fashion.

Various modules can be collapsed into a subtree, equivalent to the different levels of the data flow diagram. A subtree allows the structure chart to contain fewer objects, simplifying the amount of information the programmer has to absorb. By clicking on a subtree, the various modules in that subtree can be displayed.

As with the previous tools, a syntax checker allows the programmer to verify that the information in the structure chart is properly formatted. A further checking option allows a check of completeness that makes sure the structure chart, module specifications, and data dictionary all contain the necessary information represented graphically.



TITLE: Read_Miles_Since_Oil_Change(Miles_Since_Oil_Change)

PARAMETERS:
Miles_Since_Oil_Change : DATA_OUT

LOCALS:

GLOBALS:

BODY:

```
Read Miles_Since_Oil_Change From Global Store.

RETURN(Miles_Since_Oil_Change)
```

Courtesy of CADRE

Fig. 11-6 Structure Chart with a Module Specification

Extending team*work*

The team*work* tools represent a fairly complete set of modeling and definition tools for designing the logical representation of an information system. One of the purposes of the Relational Technology/CADRE alliance is to extend this software engineering environment beyond the design phase into the implementation and testing phases. One important extension allows the entity relationship model of an information system to be graphically translated into a physical representation in the database.

Team*work* has several facilities that allow the current environment to be extended. New tools can be defined and added to the menu of available tools. These tools are also able to access the team*work* data dictionary, allowing the use of information previously defined in other modeling environments.

Extending team*work* is done using a tool kit that allows the developer to access various portions of the existing environment, such as the data dictionary. The developer needs to do at least two things: integrate the new tool into the team*work* menus and access the data dictionary.

Adding a new application to the main menu bar of team*work* is fairly simple. The user edits a file called config_file and adds an entry that contains the name of the new menu bar and the location of a file that has the options available on the menu.

The menu file then contains information on each of the components of that menu. Each menu action has a name, which appears on the pull-down menu. It also has associated with it an action—typically a program to run. More complex forms of menus allow the user to pass a paste buffer into the new tool, or invoke different editors.

Accessing the data dictionary is done through a library of functions that allow low-level access to the elements stored in the dictionary. A variety of predefined functions are also supplied that perform higher-level operations, such as counting the number of processes in a model or allowing the programmer to attach a text note to an object. Team*work* thus provides an open environment, allowing different types of users to customize the basic tools for specific types of applications. These basic interface methods are being used by Relational Technology and CADRE to develop a variety of extensions to the basic models.

The first extension being developed ties the models developed in the three basic tools to a database system. The entity relationship diagrams and data stores can be implemented as INGRES tables. This logical to physical design tool allows indices, views, storage structures, and other components of the physical design to be specified. The specification of the physical structure, like the other editors, is done graphically. Instead of issuing a series of SQL statements, the user can graphically implement the physical design. The design editor then issues the appropriate SQL calls and constructs the actual database. This is in sharp contrast to present development methods that take a logical design and require the programmer to manually translate it into a physical design using the data definition language (DDL) commands in SQL.

The initial development phase thus concentrates on the data stores in an information system. The other parts of an information system are the modules and processes that operate on those data stores. A long-term plan is to allow rapid translation of process specifications into INGRES 4GL code. With a graphical INGRES 4GL generator tied to the analysis tools, as well as a logical to physical design translator, it is possible to begin rapidly prototyping information systems. The designer could take a model and have a variety of forms, reports, tables, and INGRES 4GL procedures automatically generated. If a process is not fully specified, a simple "return" could be substituted for the INGRES 4GL code.

The implication of this joining of the INGRES development tools to the team*work* analysis tools is that structured development and rapid prototyping methodologies can be at last brought together. In most CASE environments, the design phase proceeds without the benefit of seeing what the system will look like. This results in internally consistent designs that may not reflect reality. Rapid prototyping allows both designers and users to quickly see the implications of various design trade-offs.

Logical and Physical Database Design

All three of the tools examined in this chapter—entity-relationship diagrams, data flow diagrams, and system design structure charts—allow the software developer to construct a logical model of the information system. These logical models need to be translated into physical designs as they are implemented.

There are two aspects to the physical design. First, the data stores, entities, relationships, and other concepts need to be translated into their equivalent logical database structures of tables and columns. Next, the logical database design needs to be turned into physical database structures such as indices, permits, and integrities.

For the logical model to be a physical design translation, Relational Technology and CADRE are both working on extending the team*work* environment to aid software developers in this process. This enhancement of team*work* begins by providing a physical design tool and will be supplemented next with a logical design tool.

Logical Design

The process of a database logical design entails translating entities and relationships into database tables. Take, for example, a one-to-one relationship between two entities. This could entail the fact that each employee has a single office and each office has a single employee in it. In this case, the translation from the entity-relationship model to the relational database model is fairly simple. An employee table is constructed and that table has columns for both employee and location.

For one-to-many relationships, the process is slightly more complicated. If a salesperson has several sales outstanding, this is a one-to-many relationship. In this case, two tables are constructed. One table is for the entity salesperson. This table has a unique key for the salesperson, say the last name.

The sales table needs to have one line for each sales order. It also needs to have the name of the salesperson. The salesperson name is the same name as is in the other table and is known as a foreign key. By joining the two tables together, the user can find out information about the relationship. Some information in the entity-relationship model cannot be reflected in a relational database. For example, the cardinality of a relationship cannot be directly reflected in a design consisting of a series of tables. Instead, the cardinality aspect of a relationship would have to be implemented as an integrity constraint that regulates the input and update of information in one or more tables.

Translating an entity-relationship model into a relational database logical design usually consists of two steps. First, the formal method of normalization is applied to the entities and relationships to break them down into a series of tables. Normalization ensures that updates to the database can be performed in a consistent fashion and that data are not overly redundant.

The next step is to change the logical design to reflect the nature of the applications that will use that data. Normalization, by reducing redundancy, also makes retrieval of data more difficult because many small tables may have to be joined together. Based on the types of applications that will use the data, the designer may add redundancy or otherwise violate the rules of normalization.

Normalization

Normalization is a formal method that attempts to solve the problem of update anomalies. An anomaly results when data are stored twice and are only updated once. It should be stressed that normalization is only one of several database design techniques. Although normalization reduces update anomalies to data, it does often make retrievals more inefficient by breaking data up into many different tables.

The process of normalization consists of breaking tables down into smaller tables. There are various levels of normalization. This book contains a brief discussion of the first three normal forms. There are also fourth and fifth normal forms available and research continues on further extensions of this technique. The reader is referred to C. J. Date, *An Introduction to Database Systems,* Volume I (4th ed., Addison Wesley, 1986, Reading, Mass.) for a more formal treatment of this technique.

First normal form consists of eliminating repeating groups. A repeating group means that a table has several columns, each of which represents the same piece of data. For example, an employee might have multiple phone numbers. First normal form is violated when there are two columns, one for each phone number. The update anomaly results when a user attempts to update a phone number. The application program would have to search both columns to find the relevant phone number. In a design conforming to first normal form, employee information would be broken into two tables. The employee table would have one row for each employee. The phone table would consist of two columns, one for employee name and the other for the phone number.

A relation is in second normal form if every nonkey column is dependent, directly or indirectly, on the key for that table. For example, a projects table could have as the primary key the columns employee and project name. Storing a department's budget in the same table would be a violation of second normal form because the budget is not dependent on either portion of the key.

Third normal form states that any nonkey column in a table must be dependent on the whole key for the table. Using the same projects table, storing employee office locations would be a violation of third normal form because the office location is only dependent on a portion of the key—the employee name and not the project name.

To achieve normalization, the table would be broken up into two tables. One would have project-specific information, such as the number of hours worked by the employee. The second table would contain only employee-specific information such as office location. The update anomaly in a violation of third normal form exists because the employee office location exists once for each project that the employee participates in. If the user updates the office location, he would have to ensure that every instance of the phone number is updated.

As can be seen, the process of normalization consists of breaking a table down into a series of tables, each containing a well-defined group of information. The rigorous process of normalization helps the database designer identify potential problems in the design and some ways of curing those problems.

Unnormalization

As can be seen, normalization can quickly lead to a large number of small tables. Every time the employee office location, telephone number, and projects are retrieved, three tables would have to be joined. For this reason, the database designer often violates various normal forms in the process of designing the database. The implication of violating these rules is that updates become more complex, with the programmer having to make sure that all the occurrences of the data are updated.

As a general rule, violating normal forms trades off update efficiency for quick retrieval of data. Often, this is done by adding controlled redundancy. The first technique is to join tables together in violation of the normal forms. The second technique is to add derived data to the database for information that is frequently retrieved.

Prejoins are one technique used to increase the efficiency of retrievals. Normalization might dictate that two tables should be stored separately, even if there is a one-to-one mapping between the two tables.. However, if users are always performing a join on the two tables, this doubles the number of I/O operations, not to mention making it harder on the programmers and users that have to access the data.

Another database design technique for increasing performance is when there is an almost one-to-one mapping between two tables. Good database design techniques might dictate that these two entities be stored in separate tables. An example of this situation is families and homes. In almost all cases, families have one home. There are a few exceptions. To manage this situation, it is possible to keep the primary home for a family in the family table. Then, a flag field is added to signal the presence of an exception. This exception management technique adds a little redundancy, but it is able to handle the majority of the cases simply.

The downside of this technique is that all applications need some special code for exception handling, particularly in the case where normal and exception data must be treated together as a union. Horizontal partitioning is especially difficult for people browsing the database, such as a QBF user, because there is no intuitive manner of seeing from a list of tables just how they are related.

Another violation of the normalization techniques is to introduce repeating groups into a table. For example, there are two ways of storing sales data. One method has a series of columns, one for each month of the year:

```
sales_table ( salesperson, january, february, march ... )
```

The second method, using normalization techniques, would have only three columns:

```
sales_table ( salesperson, month, units )
```

The first form is easier for crosstab reports that show the months of the year going across the page. The first form also has the advantage of making sorted retrieval since only the salesperson column and not the month column have to be sorted. On the other hand, certain queries, such as showing the month with the greatest sales, involve a large number of "or" clauses in the query. It is possible to keep both tables in the database, although application programs have to make sure to update both versions of the data.

Vertical and horizontal partitioning is another way of increasing performance. Normalization might allow an address and a resume field to share the same table. However, people frequently access addresses and infrequently access resumes. To increase performance, it might make sense to vertically partition this table into two pieces. Each table can then have different storage structures. More importantly, people that want to access addresses do not have to retrieve the resume.

Horizontal partitioning breaks a table into two pieces based on the values in a row. For example, historical data might be moved into a separate archive table. Normally, users would only access current data. Since there are fewer rows in the current table, horizontal partitioning increases efficiency. However, when both historical and current data are needed, the query needs to search both tables.

The second class of techniques for increasing efficiency is to add data to the database that are artificially generated. An example of artificially generated data is summary sales information that is based on some aggregate of the basic sales data. Three examples of artificially generated data are:

- calculated columns

- aggregate columns
- sequential keys

A calculated column or aggregate column is one that depends on data in the database. A calculated column is one that is derived from one or more other columns in the table. For example, a table might store salary and commission information for a salesperson. A calculated column called total compensation could be added to the table defined as the sum of the salary and commission columns.

Adding a calculated column increases retrieval speed if users are often rederiving this information. However, whenever a salary or commission is changed, the application program must make sure to update the total compensation column. Although a calculated column is based on an individual row of data, an aggregate column spans several rows. For example, a frequent piece of aggregate information for sales data would be the sum of sales by sales district. Rather than rederive this information every time it is requested, the database designer could include a separate table that kept summary sales information.

Note that when new sales figures are entered, the aggregates are out of date and must be refreshed. An alternative to storing the information as a separate table is to define a view with the same information, although this requires the information to be recalculated every time it is retrieved.

Sequential keys, another form of artificial data, are keys assigned by the application programmer. An example would be a purchase order database. If all customers are allowed to give their own purchase order numbers, the data in the database could have a wide variety of different formats depending on the customer. A sequential key would assign a unique number to each new purchase order and would be the primary key for the table. A secondary key for the table would be the original purchase order number.

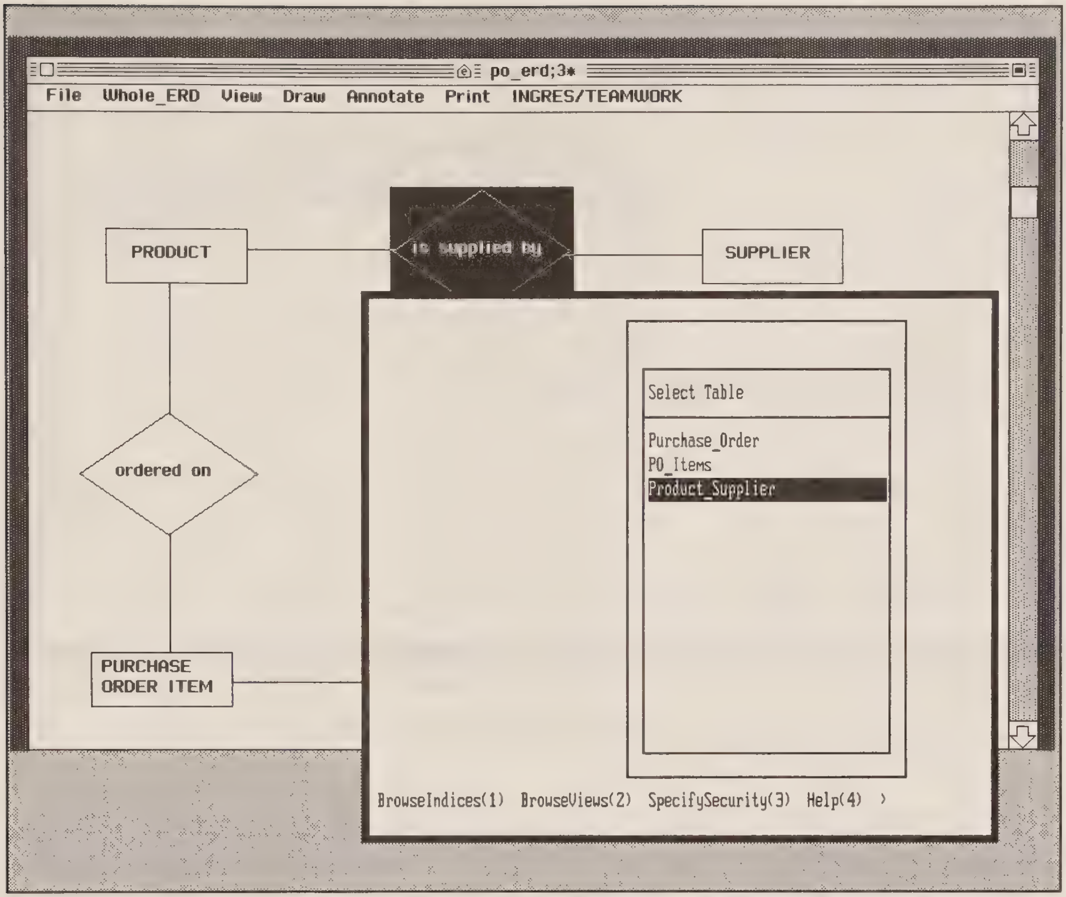
Sequential keys are a subclass of a group of keys called surrogate keys. A surrogate key is any key value for a database table that is artificially generated. Often, these surrogate keys are sequential in nature, but there are exceptions.

An easy method to keep track of sequential keys is to keep a separate table in the database containing the largest current key value. When a new purchase order is entered into the database, the programmer retrieves the current value of the table and increments that table by one.

This technique has two important benefits. First, when a new purchase order is entered, it is not necessary to scan the entire base table to find the current maximum key value. Scanning the entire table would of course prevent any other users from writing data because the scan would require a shared read lock on the entire table.

In addition to increasing concurrency, this technique also reduces the number of I/O operations required. Instead of performing a relatively inefficient aggregate on the data, the programmer goes straight to the one-row table containing the current maximum key.

Constructing a logical design for a database is a difficult process, involving tradeoffs between the different types of applications that will access the database. There are no hard-and-fast rules for a logical design. The logical design is extremely important,



Courtesy of CADRE and Relational Technology

Fig. 11-7 Translating an ERD into a Physical Design

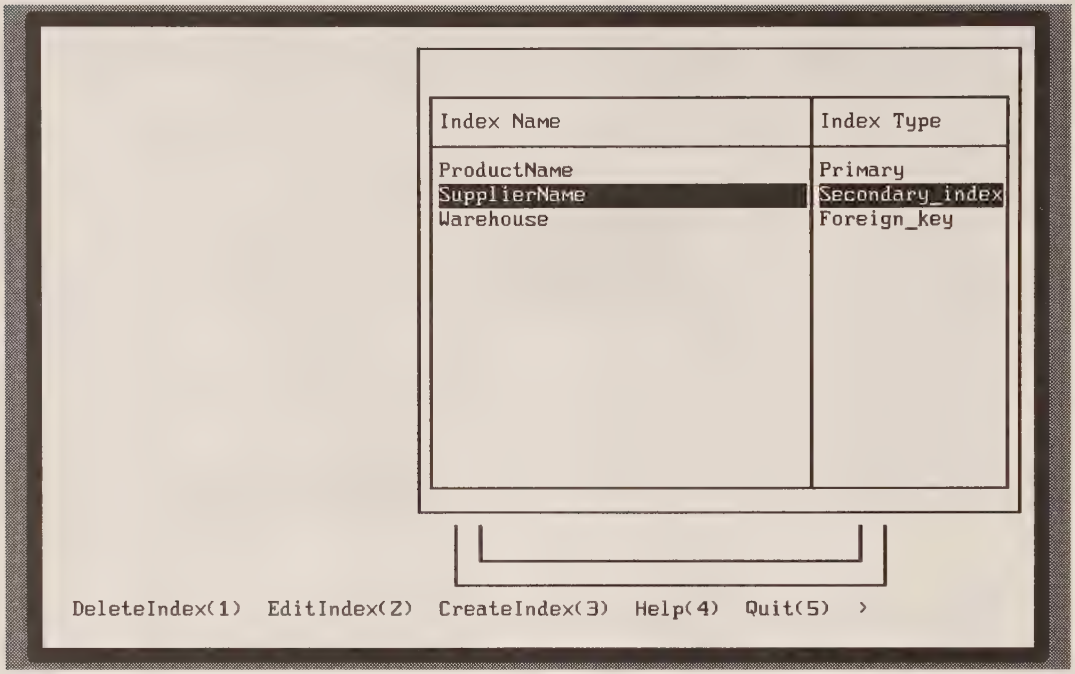
however, because the application programmers write SQL statements that use the tables formulated in the logical design process.

Logical to Physical Design Translation

One of the most important decisions for the database designer is how to translate a logical database design consisting merely of tables into the more physical representation of that design in the database. This step involves a variety of issues, such as designating primary and secondary keys, specifying security, and constructing views on the data.

The team*work* physical design tool allows a user to specify physical design criteria for model elements. For example, Figure 11-7 shows the physical design tool being used on an entity-relationship diagram. The *is_supplied_by* relationship has been highlighted and the physical design menu option selected.

The physical design tool lets a user select an existing table or construct a new one. In addition, the user has options to construct or edit indices, views, or security informa-



Index Name	Index Type
ProductName	Primary
SupplierName	Secondary_index
Warehouse	Foreign_key

Courtesy of CADRE and Relational Technology

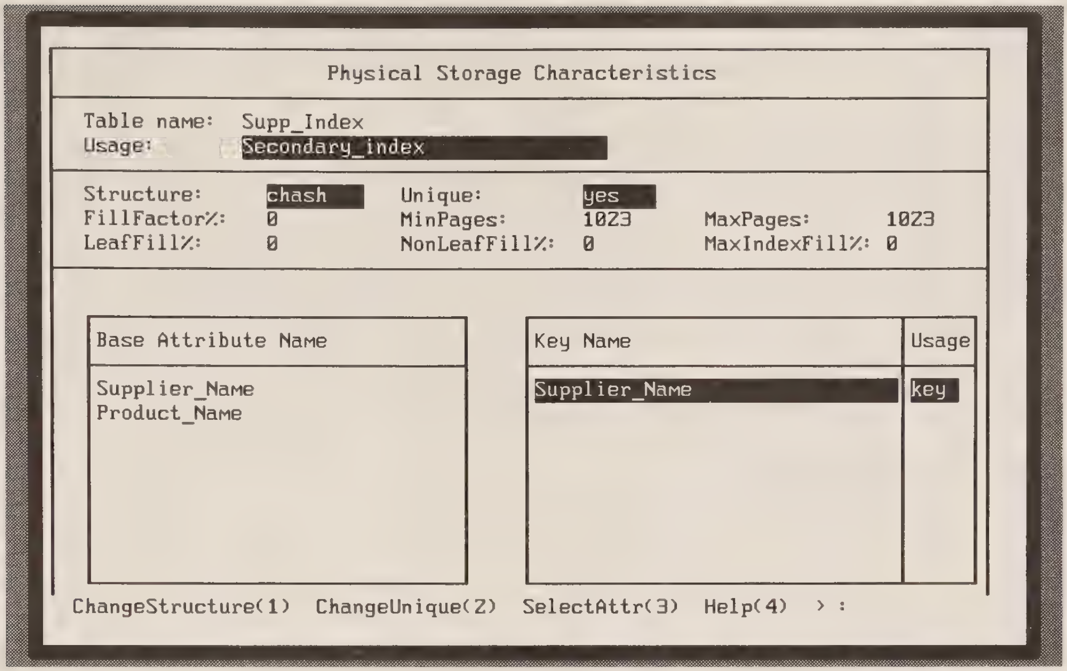
Fig. 11-8 Creating Indices for a Table

tion. Once a table has been constructed, the design tools let the user specify the storage structures of that table. For example, in Figure 11-8, the user has highlighted the *Supplier Name* secondary index and picked the edit index option. Figure 11-9 shows the screen that allows the user to change the physical characteristics of the index.

The physical design tool allows a user to translate the abstract concepts on the team*work* model into an INGRES database. The user can construct tables, specify storage structures, construct views, and designate columns as foreign, primary, or secondary keys. The user can also specify security on the various columns and tables of the database.

The next step in the Relational Technology/CADRE development effort is a logical design tool that allows the user to quickly translate the entity relationship diagram or data flow diagram into a logical design consisting of tables and columns. The user can also specify information such as the usage patterns for particular columns. A column can be flagged as having a high update frequency, for example. This information would be used when the logical design is turned into a physical design. A high-update column, for example, would not necessarily be a good column to be a key for a table because this would lead to concurrency problems.

A column in the database with a high retrieve frequency, on the other hand, would be a good candidate to be a primary or secondary key for a table. The physical design could also include information on gathering statistics using OPTIMIZEDB on that particular column.



```
                    Physical Storage Characteristics

 Table name:  Supp_Index
 Usage:       Secondary_index

 Structure:     chash      Unique:        yes
 FillFactor%:   0          MinPages:      1023     MaxPages:       1023
 LeafFill%:     0          NonLeafFill%:  0        MaxIndexFill%:  0

  Base Attribute Name              Key Name                     Usage

  Supplier_Name                    Supplier_Name                key
  Product_Name

 ChangeStructure(1)  ChangeUnique(2)  SelectAttr(3)  Help(4)  > :
```

Courtesy of CADRE and Relational Technology

Fig. 11-9 Setting Physical Characteristics

The third stage of development is a tool to translate process specifications into INGRES 4GL code, forms, and the other components of an application. When all three stages of the development are completed, the designer will be able to take an abstract model of an information system and quickly generate an implementation consisting of a prototype ABF application and an INGRES database.

Summary

This chapter examined three traditional CASE tools used for the development of information systems. Entity-relationship diagrams (information modeling), data flow diagrams, and structure charts are different models of the information system used for different stages of the design and development of that system.

Because INGRES/team*work* is an open environment, it can be extended for methodologies used in structured development. For example, other graphics-based editors can be developed that conform to other views of how to go about designing the specifications for an information system.

An important extension to team*work* is the inclusion of tools to aid in the design of database systems. These tools allow the developer of a model to quickly design an INGRES database. A model is translated into the database logical concepts of tables and columns.

Part of the process of constructing the logical database design is normalizing the tables to prevent update anomalies. Just as important is unnormalizing the tables to increase efficiency for retrieval operations.

Next, the tables and columns (the logical design) are turned into a physical design, consisting of specifying storage structures and other aspects of the database that the front-end system are not aware of, but that increase performance in the system.

A further extension of team*work*, envisioned for the future, allows process specifications to be quickly turned into INGRES 4GL code. This would allow both the data repository and the applications that work on that data repository to be quickly turned into working prototypes.

Chapter

12

Tools for Building an Information Architecture

Architectures

Managing application development and data is a particular challenge given the current pace of change in computers, software, networks, and user requirements. An information architecture structures the decisions on how to acquire and use components in the computing environment in a way that preserves the investment in software and training and permits a smooth migration to new technologies. A well-specified information architecture gives an organization the ability to change one component of the computing environment without being forced to change other components simultaneously.

An architecture is essentially a specification of the interfaces between different components of a system. For example, SQL is an interface between an application and a data manager. The interface is carefully defined, allowing a change in one of the components without changing the other.

The importance of an interface such as SQL can be shown in the development of the Simplify tool set. The Simplify tool set supplements the terminal-based INGRES utilities with workstation-based graphics tools. The Simplify developers were able to concentrate on changing the nature of the user interface without worrying about the impact on the back-end data manager. As long as Simplify produces SQL, the data manager is able to service the request for data.

An information architecture is a collection of key interfaces such as SQL. The purpose is to allow users to find and access information in a distributed, heterogeneous network characterized by a variety of data managers, computing platforms, and network protocols and to allow a smooth migration to new technologies without requiring a restructuring of all portions of the computing environment.

The information architecture for an organization is a planning tool that allows one group in the organization, such as application developers, to accomplish its job without worrying about the effect on all other portions of the organization. A new application, for example, can be developed without having to necessarily buy a new workstation for users or a new DBMS.

This chapter discusses three types of issues. First, it contains a brief description of what a distributed, heterogeneous computing environment might look like. This discussion shows the nature of the changes that many organizations are experiencing. Note that the picture of a computing environment presented in this section is only one of many possible pictures that can be drawn. It is presented only as a tool for discussing how a particular information architecture might be developed.

Next, the chapter discusses the key interfaces and components that make up an information architecture and the relationship of the INGRES tools to the architecture. Finally, the chapter discusses how an organization might go about defining and using the key interfaces that make up the information architecture.

Change in the Computing Environment

Change is a fact of life for all aspects of a computing environment. New computers are developed with more power, allowing new software systems and applications to be developed. User requirements change and the computing environment must also change to meet the new user requirements.

One can debate endlessly about which came first—new user requirements or more powerful computers. While this chicken versus egg debate is fascinating, it does not matter for purposes of this book. Instead, the important point is to keep the two in synchronization. Computers should match the information needs of users, and users should be aware of the capabilities of new systems.

Three areas of change are discussed in turn:

- hardware platforms
- networks
- database management software

Change in Hardware Platforms

Any attempt to plan for change has to take into account the rapid increase in the power of computers. Many organizations are forced to frequently upgrade hardware platforms in order to run software that only runs effectively on these more powerful machines. An example can easily be found in the early personal computers. It was not unusual to find personal computers with 64 to 256 Kbytes of memory, and users that could not conceive of why they would want to upgrade to 640 Kbytes of memory.

	CATEGORY	1984	1987	1990
DEPARTMENT	PROCESSOR	VAX 11/780	VAX 8800	ELXSI 6400
	MEMORY	8 MB	128 MB	1-4 GB
	DISK	500 MB	4 GB	100-500 GB
	COMMUNICATIONS	56 kbps	10 mbps	100 mbps
	MIPS	1 MIP	10 MIPS	100 MIPS
DESKTOP	PROCESSOR	IBM PC/XT	MicroVAX II	Sun 4
	MEMORY	256 KB	8 MB	32 MB
	DISK	10 MB	159 MB	500 MB
	COMMUNICATIONS	300 bps	10 mbps	100 mbps
	MIPS		1 MIP	10 MIPS

Fig. 12-1 Migration of Computing Power to the Desktop

Of course, when those users obtained copies of Lotus 1-2-3, they quickly realized that 256 Kbytes was not at all sufficient. Lotus did not function effectively in that environment. If users wished to take advantage of the simple user interface of Lotus, they needed a more powerful hardware platform.

Desktop publishing and CAD/CAM are two more examples of the need for more powerful computer systems. In the early days of workstations, 1 Mbyte of memory was considered to be a standard configuration. In many cases, 2 Mbytes was considered to be highly extravagant. Today, many workstations cannot be purchased without 4 Mbytes of memory, and many users insist on having 8 Mbytes or more of memory.

The increase in capacity of these computer systems consists of increases in many different aspects of the computer (see Fig. 12-1). The CPU is getting more powerful. What used to be considered a powerful departmental computer, the 1 MIP VAX 11/780, is now considered a cheap personal workstation in the form of the MicroVAX II. Current departmental computers in the form of a 10-MIP computer are rapidly showing up on desktops. Higher-end systems such as 50- to 100-MIP computers are also beginning to show up on desktops. Note that the computers used in Figure 12-1 are typical examples; many different brands of computers are available in each of the categories.

Accompanying the move of CPU power to the desktop are the other portions of the computing systems; 8 Mbytes of memory used to be considered a large multiuser configuration. This is now a standard desktop configuration. Large systems are increasingly being configured with hundreds of megabytes of main memory.

Secondary storage, in the form of a 500 Mbyte disk drive, was once considered a standard departmental configuration. Most large systems now have several gigabytes of disk space, and many personal workstations have hundreds of megabytes of storage. Tertiary storage in the form of optical disk and other media are rapidly increasing the amount of on-line data available on a computer system.

Larger systems are being used for more sophisticated software, as well as the solution of more complex software. As the cost of hardware decreases and the cost of people increases, the decision to buy more powerful systems becomes an attractive one

Distributed Processing

While computers are becoming more powerful, they are also becoming more specialized. Instead of running different programs on a single computer, many networks consist of a large variety of specialized servers, each dedicated to a specific task.

Figure 12-2 shows a possible configuration for a distributed network. The access point to the network is a series of workstations dedicated to providing the user interface. These workstations use a windowing system such as the X Windows System, IBM's Presentation Manager, or Apple's Mac Toolkit. The windowing system allows applications across the network to all use the display on the user's workstation. The next level of the network is thus a series of application servers. Data browsers, statistical systems, and desktop publishing are just a few of the applications that could be present on the servers.

Two other types of servers could be present at this level of the network. Gateways allow other networks to be accessed from the workstation. The gateway could be to a specific machine, as in the case of a gateway to an IBM mainframe, or a gateway to wide area communications facilities, such as an X.25 or ISDN network. The other servers give access to output devices such as printers or plotters.

The next layer of the network is the distributed database server. Applications all need access to some form of data, and the distributed database server provides transparent access to the various data repositories in the network. The gateway capabilities of a DBMS allows all of the heterogeneous data sources that form the distributed database to be accessed transparently.

The last layer is the servers that access data. The information can be structured as a simple file, or can be a relational or hierarchical database management system. Each data repository could be directly accessed, or can form part of a distributed database.

Notice that this picture of a network is highly distributed, consisting of a series of servers, each optimized for the specific task for which it is configured. The optimization can be in terms of the underlying hardware configuration; a database server, for example, can be configured for very large I/O requirements. A terminal server can be optimized in hardware for a very large number of interrupts caused by users hitting the keys on a terminal.

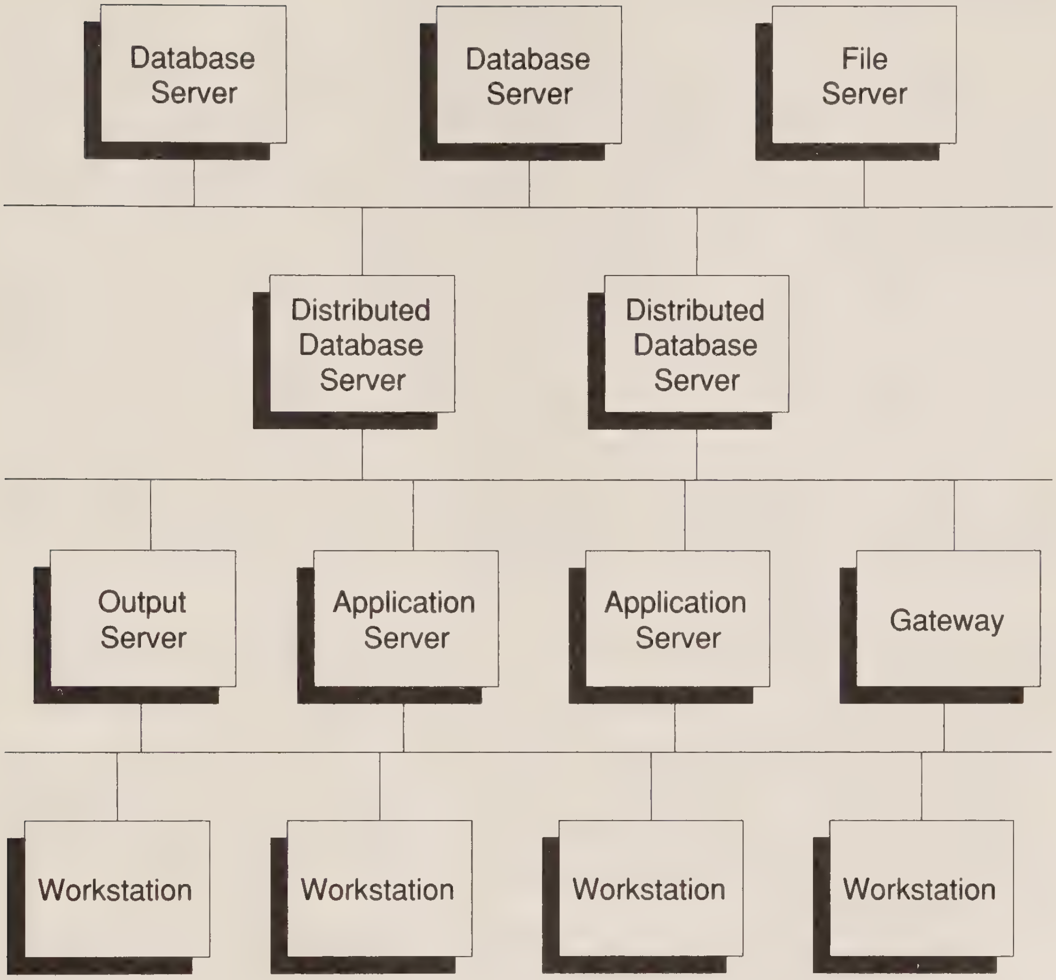


Fig. 12-2 A Distributed Network

A server can also be optimized in software. A general-purpose MicroVAX, with the VMS operating system, can be tuned for a particular type of application. If both word processing and database management were to run on the same computer, the operating system would have to be tuned as a compromise between the conflicting requirements of the two software environments.

Change in Database Software

The last key area of change discussed here is the rapid advances in DBMSs, which can be thought of as going through five stages of evolution (see Fig. 12-3). The original relational database systems, such as the university-based version of INGRES, were an attempt to allow access to data using a nonprocedural query language. Users were able to perform query language statements to retrieve information from the database and to retrieve the definition of data in the database.

The original university version of INGRES consisted of a relational database engine with one method of access—direct entry of statements using the QUEL query language. Although this provides flexibility, it is not necessarily convenient to teach managers how to program in QUEL. To do so is equivalent to the early days of personal computers, when managers were forced to write programs in BASIC to solve their needs.

The next stage in the evolution of relational databases was to add more convenient user interfaces. In INGRES, this consisted of the report writer and the forms-based query environment. When INGRES was turned into a commercial product, a program interface based on embedded QUEL statements was added.

The original embedded QUEL programs and QBF provided a fairly primitive access to data. These original capabilities were supplanted by a fully functional fourth-generation language—INGRES 4GL, which can be used by itself in the ABF environment, or a variant of it can be embedded into a traditional third-generation language.

Fourth-generation languages such as INGRES 4GL allow the programmer to focus on the business problem at hand instead of the details of implementation. The fourth-generation language increases programmer productivity in a variety of ways, including:

- providing a built-in report writer for producing most reports instead of using a conventional 3GL
- providing a forms system for rapid development of the user interface
- providing embedded database constructs to allow simple commands to solve complex data requests

Systems like INGRES tie all of these tools together using an integrated application generator that is able to use native subsystems such as QBF. The application generators allow very large systems to be quickly put together using a series of building blocks.

The next stage in the development of database systems has been performance. One of the original arguments against relational database systems was poor performance. Often, this was the result of poor database design, but the end result was a slow-running system.

Part II of this book examined a variety of different methods being used to provide high performance in a relational environment. A multi-server architecture, for example, allows many computers to all process requests for data. Increases in the intelligence of the query optimizer is another example of the increases in performance.

A few vendors, including Relational Technology, have entered the fourth stage in the development of relational systems: distribution of data in a networked environment. Part III of this book examined the General Communications Facility, INGRES/NET,

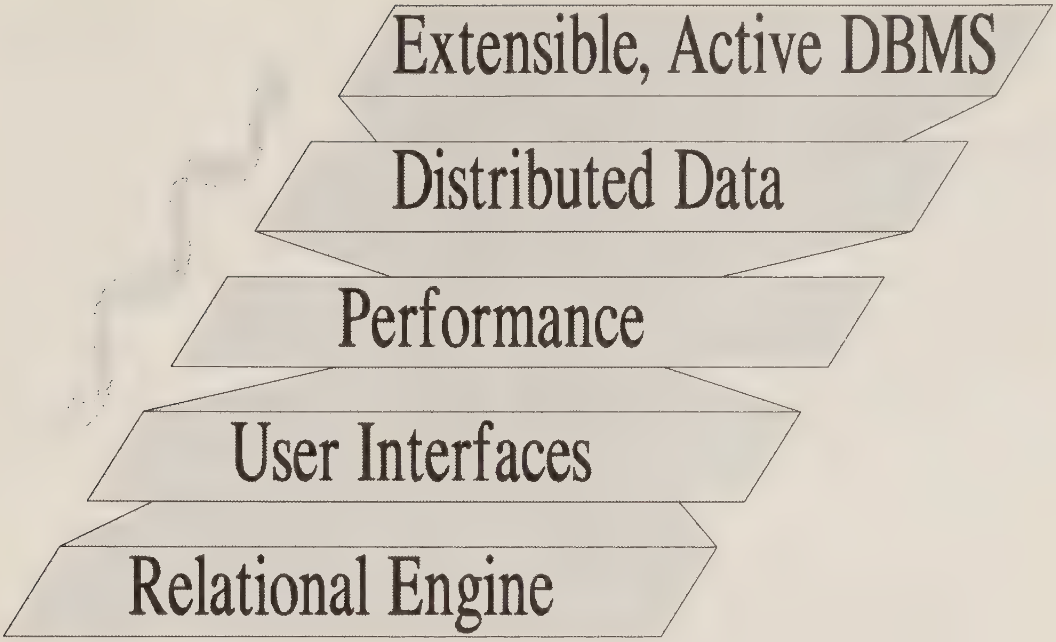


Fig. 12-3 Evolution of Relational Systems

INGRES/STAR, and gateways as the methods used to access data stored in a variety of locations and formats.

The chapter on Postgres showed what the fifth stage is in the evolution of relational databases: extensibility and active database systems. The Postgres system allows the user to define new data types, operators, aggregates, and access methods for storing and retrieving data. This extensibility allows the database to be used to store a variety of different information types, not just traditional business-oriented data.

An active database system means that the database manager can respond efficiently to very complex information requirements. The rules system in Postgres, for example, allows the database system to respond to changes in the structure of the database and inform applications about those changes.

Key Interfaces

A rapidly changing environment such as the one discussed above leads to the possibility of two forms of incompatibility:

- incompatibility among the different subsystems
- incompatibility over time

Given a network with several brands of computers, different network technologies and a variety of database managers, there is a potential for different islands of information. Users of one subset of computers and software can access some information on the network, but cannot access other repositories of information. Some applications run on certain pieces of hardware, but not on others.

The second type of incompatibility is incompatibility over time. When computers are upgraded, for example, the application may have to be rewritten. Portability of data and applications to new environments is essential for a smooth migration. Converting data and rewriting applications can increase the amount of time required for an upgrade by orders of magnitude.

Throughout this book, we have examined a variety of key interfaces that ensure the compatibility of different subsystems within a computing environment and as the environment changes. Several of these key interfaces included:

- interface between front and back ends
- interface to the underlying network
- The front-end interfaces (The 4GL and user interface)
- The back-end interfaces to the file and operating systems
- interfaces for the definition of data and applications

Front- and Back-End Communication

In the INGRES environment, the front- and back-end interface consisted of two components. First, the SQL query language is a standard method of requesting data and submitting commands. As was seen, SQL can be used to access information from any INGRES database or gateway. The second component is the GCA facility. GCA is a standard method of sending messages between INGRES components. GCA consists of a series of messages, some of which may contain SQL statements. Other messages are used to describe data, return data, or initiate sessions.

GCA messages are transmitted using either the local interprocess communication facilities or GCF. GCF provides the interface to the underlying network, shielding applications and servers from the complexities of a heterogeneous networking environment.

GCF and GCA are two very key interfaces for the application programmer. New applications can be designed without worrying about the type of data manager or the location of that data manager. Because applications are shielded from the location and type of the back end, data can be moved without rewriting applications.

The separation of the front end and the back end allows a graceful migration path for both components. A database can be initially developed in one environment, as in the case of an application that uses a gateway to access an IMS database. Later, the data can be moved over to an INGRES database without having to rewrite the application.

The Network

Although GCF shields the information architecture from the complexities of the underlying network, it is important to understand the design of an organization's network architecture. The network architecture provides the basic capabilities for communication of information. Since the database applications have the potential to move a great deal of information over the network, there needs to be enough capacity on the underlying network.

Designing a network architecture is beyond the scope of this book. Here we concentrate on a limited subset of the problem—access to data in a distributed network. The designer of the information architecture needs to be aware of several aspects of the underlying network architecture:

- support for physical and data links
- support for transport interfaces
- support for upper layer services

The physical and data links available in a network influence the possible range of servers that can be connected to that network. When purchasing a new database server, for example, it is important to understand how this machine will physically connect to the network. If the local area network is based on Ethernet, for example, the data repository needs to support a connection to the Ethernet cable and to support the Ethernet data link protocols.

The physical and data link layers govern how a particular node can connect to a subnetwork. Ethernet is an example of such a subnetwork. If both workstations and servers are located on the same subnetwork, the user needs to be assured that there is enough bandwidth for the transmission of data and compatible upper-level protocols.

The network can consist of a series of subnetworks. If a distributed database will access information across different subnetworks, it is important to examine if there is a communications path between the various nodes involved. For example, a workstation might be on an Ethernet using the TCP/IP protocols and a data repository might be an IMS database on an IBM mainframe using the SNA network protocols.

In order to connect the TCP/IP and SNA networking environments, there needs to be a gateway that connects the two. Gateways, like any computer, have a limited capacity to process information. If the database application will move a significant amount of data, it is possible that the capabilities of an existing gateway might be reached.

The transport interface is the key component for the support of GCF. As we saw, GCF allows INGRES gateways and servers to use the services for a heterogeneous networking environment. At each end of the networking environment, there needs to be a transport layer interface that GCF supports, such as the DECnet End Communications Layer, or SNA's LU0 or LU6.2 interfaces.

Finally, emerging standards for Remote Data Access (RDA) need to be taken into account. The RDA standards are an attempt to allow SQL-based applications to access any RDA-compliant data manager. A related standard, the Information Resources Dictionary System (IRDS), allows data dictionary information to be transferred among different data repositories.

These networking standards are important because they will provide, when fully defined and implemented, a flexible method for different types of systems to coexist. The INGRES/Gateways are a currently available method for accessing a heterogeneous data repository. In the future, as vendors support RDA, a general-purpose RDA gateway can be constructed by Relational Technology instead of the current case-by-case gateways. Note that this is a long-term goal, but it is important for users to track developments in this area.

Front Ends

Two key sets of concern exist in the development of front-end systems. First, the architect needs to look at standards for the display of information, consisting of windowing and look and feel standards. Next, the architect needs to look at methods for designing user interfaces: the fourth-generation language.

The windowing system governs how an application is able to access a particular workstation across the network. The windowing system allows the application to open a window, display graphics and menus, and receive input from the keyboard or pointing device.

The look and feel standard governs what an application will look like on the display. The window system provides the mechanism for the display of data, the look and feel standard is a policy on exactly how information should be presented. For example, the look and feel standard will govern what a menu looks like and the method used by the user to select a menu option.

A look and feel standard provides a common interface for the user across different tools. Since all applications operate in a similar manner, user training can be minimized. Instead of relearning the mechanics of an application, the user can focus on the substance of how to extract useful information from an application.

The fourth-generation language is the tool used by application developers to quickly develop new applications. A language like INGRES 4GL allows an application to be developed that will run on a variety of different hardware platforms. The importance of a language that can construct applications for different platforms is crucial when many different computers exist in the network. INGRES 4GL allows an application to be developed on one brand of computer and then easily moved over to another. The ideal situation for an organization would be a single fourth-generation language that is able to develop applications that conform to a variety of different look and feel standards and are compatible with a variety of different windowing systems.

Front-end standards provide independence for both the user and the application developer. Applications can be developed without a great deal of attention to the particular application server or user workstation that will be used. The developer is thus shielded from the deployment issues of application development. The user interface standards shield the user from changes in applications. The user can move to a new workstation or use a new application and still see a familiar environment.

In the ideal situation, the architect has complete freedom to move users to new workstations and applications to new computing environments. Since this ideal may not be met in reality, the architect needs to be aware of what options will be available by a particular configuration decision. For example, when a fourth-generation language is selected, that decision may limit what workstations or application servers can be used.

Back Ends

Data managers provide an interface to the file and operating systems on a hardware platform. Through the use of GCF and gateways, the application developer is shielded from the particular hardware platform or location of a data manager. Back-end standards allow data to be moved from one computing environment to another. As in the case of application development, the database administrator should be able to move data from one location to another without worrying about the particular nature of the operating or file system.

INGRES, for example, allows a database to be easily moved from a small computer up to a larger one. A database can be prototyped on a small server. Later, when the data become broadly available, the database can be moved up to a larger environment.

Back-end interfaces thus allow the database administrator to easily move data from one area to another. The data can be moved using utilities like *copydb*, or SQL statements can be used to move data from one portion of a distributed database to another. The INGRES/Gateway capabilities allow the data to be moved from one type of data manager to another using SQL statements.

The database administrator also needs to be able to take advantage of the performance features of a hardware environment. For example, the multi-server architecture allows the administrator to easily move a database into a parallel processing or VAX Cluster environment. Multiple servers can be configured, one for each of the processors.

Note that to move from a single- to a multi-server environment requires little effort on the part of the database administrator. New servers are started, and they are registered to the name server. When applications wish to connect to a particular database, the name server will return the address of a data server that can access that database.

The interface to the file system and disk drives of a computer should also be fairly transparent to the database administrator (DBA). The DBA should be able to move a database or a portion of a database over to a new disk drive on a computer without extensive reconfiguration. If a particular disk drive crashes, the database should be able to be restored to another disk drive.

Back-end standards should thus ensure that the database administrator has the ability to move data across components of a single computer, as in the case of configuring a new server or extending the database to new disk drives. The administrator should also be able to move data across operating systems and brands of data managers.

As in the case of front-end applications, this complete freedom to move data may not be easily achieved. The information architecture needs to take into account the amount of freedom available to move information. If one particular data manager is selected, for example, the architect should be aware of how much flexibility will be available to move data within a particular brand of platform, across operating systems, or to other data managers.

Defining Information and Applications

Data dictionaries and CASE tools are the methods used to define data and applications in a computing environment. The CASE tools are used to model new data repositories and applications. The data dictionary is used to store the definition of both types of information. To be useful, the CASE environment must be able to model a variety of different subsystems. Since most users will use a variety of tools, the CASE environment needs to able to model an information system as a combination of these subsystems. For example, a user might start with an INGRES application, generate a report and then move that data into a modeling environment. When the modeling is completed, the results would be further formatted using desktop publishing tools.

Once the data are defined using CASE tools, they are stored in a data dictionary. Users need to be able to access the data dictionary to find how information is stored and what applications are available. The INGRES system catalogs, for example, are a data dictionary that allow the user to examine catalogs for the presence of database tables, reports, forms, and a variety of other objects.

Establishing an Information Architecture

Establishing an information architecture consists of identifying a key set of components that will allow consistent access to data in a heterogeneous environment. The organization attempts to anticipate the need for information and tools, and to identify components that will meet those needs.

Once an interface is chosen, a certain range of options becomes available. Within the range of options, decisions can be made without affecting other components in the computing environment. For example, with INGRES/STAR, the database administrator has the freedom to move a data manager within all of the hardware platforms supported by INGRES.

Eventually, the limits of a particular interface will be reached. For example, a database administrator may decide that the features of data manager that are not supported by INGRES/STAR are needed. The purpose of an information architecture is to anticipate and try to minimize the frequency of situations where extensive reconfiguration will be needed because the limits of an interface have been reached.

To establish an architecture, the cooperation of several players in the computing environment is necessary:

- the network manager
- computing systems managers
- database systems managers
- database administrators
- data administrators
- application developers
- users

A useful starting point is to take the current computing environment, and identify the interfaces between the different components of the information architecture. Each of these interfaces allows a range of options for growth. For example, a particular fourth-generation language allows applications to be developed on a range of operating systems and to access a range of terminals and data managers.

Next, the limits of those interfaces should be identified. System managers, for example, could identify the capacity of various computing platforms to efficiently store and access data. A particular data manager might be able to service a certain number of transactions per second on a database of a given size.

The current components and their limits constitute the organization's current information architecture. The next step is to decide what type of information architecture will be needed in the future. A database administrator might identify a need for distributed access to a certain group of data managers. A data administrator might identify a need to store meta-data for a range of different applications and data managers. Users might identify certain information processing requirements such as moving data between different subsystems on a workstation.

Given a set of needs and a current information architecture, the group can then decide if the current architecture has flexibility for change and growth. If not, a new interface can be considered and it's impact on the current environment examined. For example, an organization might decide that all data managers should use SQL as a standard query language. If some applications already exist that use another query language, there will be a cost involved in moving those applications over to the new environment.

Just as an application can outgrow the capabilities of a particular hardware platform, an information architecture can also become out of date. Establishing a fixed information architecture and leaving it in place is no more productive than designing a single information system to meet all of the needs of an organization. Instead, the information architecture should be periodically evaluated and changed. While it is in place, various groups of users have the freedom to change a component without affecting others. Eventually, a component changes enough to require a change in others. For example, an information architecture might specify that distributed database access is provided via

INGRES/STAR. An evaluation of the architecture will establish a certain level of network support needed to support the data transmission rates of the remote data access.

For a period of time, database administrators will be able to establish distributed databases, and make those available to application developers. The network will be in place to allow access to information over a heterogeneous set of protocols and a set of communications links. Eventually, the distributed databases may outgrow the capacity of the network. At that point, there will be a need to upgrade network facilities to meet the needs of the database applications.

The architecture thus consists of a set of specifications that specify a range of activities that can take place. For a period of time, the activities in one area can continue without extensive participation from other groups. When the limits of the current facilities are reached, the groups need to reevaluate the architecture and decide what resources in each area are needed to implement it.

The architecture specification is a cyclical process. It establishes planning for access to information as a key activity. The planning process is not static—it is a continual process of evaluating the current requirements and anticipated growth against the facilities available to meet those needs.

Conclusion

In the past, information systems were static in nature. The MIS department would survey user requirements and design an information system. This would be translated into a set of design requirements and the underlying hardware architecture would be purchased and the application developed.

Often, by the time the application was developed several events would occur. First, the application would take significantly longer to develop than projected. By the time the application was deployed user requirements would change. Often, the original design did not work because the performance requirements were underestimated or the design was too complex.

Another frequent occurrence was that users would write their own applications. With the advent of Lotus 1-2-3 and personal computers, it became easy to write small, special-purpose applications. These applications were in effect a small, independent information system. The central information system was not able to get data from the spreadsheet. When data migrated from the central database to the spreadsheet, it became unavailable to the rest of the community.

Structured design methodologies are an important part of the design and development of today's information systems. They are only part of the picture, however. Most computer networks consist of a large variety of different types of equipment, software, information systems, and other components. The challenge for the planner is somehow integrating these various components into a computing environment that is responsive to change and to the needs of the users.

The information architecture is an attempt to provide an environment that is responsive to the rapidly changing nature of user requirements and of the computing environment. Instead of solving the problem of a particular information system, the architecture goes one level deeper and provides an environment that will allow structured design of information systems to coexist with ongoing computing efforts.

The goal of this information architecture is to provide an environment that allows end users and application developers easy access to data repositories anywhere in the underlying network as well as a flexible environment for developing complex user interfaces. To achieve this goal, we need to look at a variety of different issues. First, both computers and database systems are constantly changing. An understanding of this change is necessary if we are to plan a flexible system that will survive the change without extensive conversion efforts.

Next, we need to understand the underlying network that the various components that deliver information run on. If a data repository lives on an IBM mainframe and a user wishes to access that data from a Sun Workstation, there needs to be a path between the two machines. Once these underlying concepts are in place, we then examine the various components of an information architecture: user interfaces, data repositories, and a coordinated development environment. The challenge for the information architect is to be able to respond to this rapidly changing environment. The key is to provide a flexible, integrated set of tools.

Flexibility allows components to be added or moved without reconfiguring the rest of the environment. It should be possible to add a new disk drive to a computer, for example, without having to inform the users of the data server. It should be easy to add a new user to the network without reconfiguring each of the computers on the network.

Flexibility allows rapid responses to particular problems. If a data server is running slowly, the decision on whether to upgrade a particular computer or move the server to a different computer can be made without having to worry about the effect on the applications. Adding a new repository can be done without having to worry about integrating the data onto an already overloaded computer. The integration of tools is what allows these decisions to be made in a modular fashion. GCF, for example, is what allows multiple data repositories to be reconfigured, and still appear as a single logical database to the users.

Providing tools for access to information shifts the focus from developing a single model of an information system. Tools are provided to develop models of complex applications, but it is realized that these complex applications will have to coexist with general-purpose tools such as Lotus 1-2-3 or QBF.

A focus on tools shifts the focus away from the application developers toward the network and the end user. The goal is to provide the end user with access to data. If the complex application is not fully developed, it is still possible to quickly prototype an ABF application to access information. If a transactions processing system does not have the necessary reports for a new requirement, the user can use RBF to provide that information.

Glossary

20/20 A spreadsheet that runs on VAX and other computers made by Access Technology.

3270 Display Stations Terminals for IBM mainframe computers.

370 architecture IBM architecture for mainframe computers, including the 3090 processors.

4GL See fourth-generation language.

3GL *Third-generation language.* Traditional programming languages such as FORTRAN.

ABF See Application-By-Forms.

abort Stop a function before its normal completion. For example, the INGRES database may abort if the computer crashes.

abstract data type A data type not native to a computing environment. Integers are native data types; dates are abstract data types because the software has to understand the format of the date.

Abstract Syntax Notation A presentation layer protocol in the ISO networking standards. Provides a way for information to be represented in a machine-independent manner.

accessdb An INGRES utility used to authorize new users or configure new location names.

access method A means of accessing information in a file. ISAM is an example of an access method.

access name A term used in the IRDS data dictionary standards. The access name is a unique name for an object (entity) in the data dictionary.

access plans Different ways of retrieving information from files that make up a database. A query optimizer will generate a variety of access plans and choose the one that it estimates to be optimal.

Access Technology Makers of the 20/20 spreadsheet.

ACCOUNTING A VMS utility used to keep track of resource utilization by users.

active database A database system, such as Postgres, that is able to respond via triggers and rules to the changing nature of the data it keeps.

ad hoc Latin phrase meaning for a specific instance. Used in computing to refer to not previously planned functions.

aggregate A function in a query language used to perform an operation on several rows of data. Sum is an example of an aggregate.

Alerters A Postgres concept. An alerter is a rule that is activated when a certain condition occurs in the database and a program is notified of the event.

ALL-IN-1 DEC's office automation shell, consisting of a menu driver, a mail user interface, a calendar manager, and a file manager.

alternate location A location is a disk drive that contains INGRES files. An alternate location allows a database to be spread over multiple disk drives.

anomalies An event that leads to inconsistencies. A database that is not properly normalized has the possibility of update anomalies when one occurrence of the data is changed and other instances of the same data are not changed.

append A query language command to add new data to a database table.

application A program that performs functions for a user. An order entry system is an example of a custom application. QBF is a general-purpose application.

Application-By-Forms The INGRES application development environment.

application generator A program used to generate other applications. ABF is an application generator because it aides the programmer in quickly developing a new application.

applications layer The top layer of the network protocol stack. The applications layer is concerned with the semantics of work. For example, getting a certain record from a file by key value on a foreign node is an application layer concern. How to represent that data or how to reach the foreign node are issues for lower layers of the network.

architecture A set of plans that allows different components to work together. A network architecture allows different computers on a network to communicate. An information architecture allows different users to access a variety of data repositories.

archiver An INGRES process that moves data out of the transactions log into a journal. The journal provides a backup in case data are corrupted in the database.

ART An artificial intelligence environment made by Inference Corporation.

ASCII *American Standard Code for Information Interchange.* A standard character set that assigns an octal sequence to each letter, number, and selected control characters. The other major encoding standard is EBCDIC.

assignment An operation in the INGRES 4GL where data is assigned to a variable such as a field on the form.

attached query An operation in the INGRES 4GL where two queries are connected. The first query is the master query. For every row in the master query, the second query (the detail) is executed.

attributes This term has a variety of meanings. In a relational database, attribute is another name for a column in a table. In a data dictionary or other information model, an attribute is attached to a relationship or entity. Number of times modified is an example of an attribute for the entity "User_Name."

auditdb An INGRES utility used to examine the journal files to determine which users performed which operations on a database.

back end A general term used to denote all the programs in a database system that get data for a user. An application is a front end and it dispatches SQL statements to a back-end data server, which in turn returns rows of data.

backloader A program used to take an existing information system and load the definition into a data dictionary.

bandwidth The amount of data that can be moved through a particular communications link. Ethernet has a bandwidth of 10 mbps.

base data Contrast with derived data. Base data are the data originally entered into the database. Derived data could be aggregates, calculated columns, or views.

BASIC *Beginner's All-purpose Symbolic Instruction Code.* A programming language.

BBN *Bolt, Beranek, and Newman.* A company that specializes in communications. Responsible for the Defense Data Network. Also makes the RS/1 data modeling software.

binary tree Often referred to as a BTREE. A storage structure with a dynamic index used for environments with frequent updates to data.

bit-mapped A graphics term in which all bits of a display station are controllable. Contrast to a character-oriented terminal.

Boyce/Codd Third Normal An relaxed version of third normal form used in database design.

bps *bits per second.*

break columns A term used in report writers for columns that are sorted.

Broadband A physical medium used for Ethernet and other local area network technologies.

BTREE See binary tree.

bubble A term used in INGRES/team*work* to refer to a process.

buffer A portion of main memory on a computer used to hold data.

buffered I/O Used in the VMS operating system to refer to terminals. Contrast with direct I/O.

bus The part of a computer that connects devices so that they may communicate. An XMI bus, for example, connects memory cards, CPU cards, and peripheral buses (the BI Bus). The BI bus allows multiple peripheral controllers to be connected.

C A programming language. Often used on the Unix operating system.

cache A portion of main memory used to cache pages read from disk. If a page of data requested is found in the cache, the program is spared from having to go to the disk to get the data.

CAD/CAM *Computer-aided design/computer-aided manufacture.* Software/ hardware combinations for the automation of engineering environments.

CADRE Makers of the team*work* CASE software.

calculated column Information in a retrieval that is derived from data in the database. If sales and quota are two database columns, sales minus quota would be a calculated column.

cardinality Used in entity-relationship diagrams. Cardinality refers to the number of instances of one entity that can or must participate in a relationship.

cartesian product The combination of every row in one database table with every row in another table.

CASE *Computer-aided software engineering.* A term used to refer to a set of tools that help automate and control programming environments. Examples of CASE tools would be INGRES/team*work* models.

catalogdb An INGRES utility used by end users to find out what databases they own.

catalogs Short-hand for system catalogs. Tables in the INGRES database used to manage itself.

cell The intersection of a row and a column in a spreadsheet or table field.

checkpoint A snapshot of the database at a point in time used for backup.

CI bus *Computer interconnect bus.* Refers to the 70-mbps bus and controllers used in the VAX Cluster. To be contrasted with Local Area VAX Clusters that use a 10-mbps Ethernet as the transport mechanism.

CL/1 *Command Language/1.* A programming language developed by Network Innovations to provide access to VAX and IBM databases from Macintosh workstations.

clustering Shorthand for data clustering. Data are clustered when similar values of data are stored close to each other on the disk. Since users typically retrieve several rows, data clustering reduces the number of I/O operations.

CMS *Conversational Monitor System.* The user interface on IBM's VM/CMS operating system.

COAX *coaxial.* A type of cable used in Ethernet networks.

COBOL *Common business-oriented language.* One of the first standardized computing languages.

CODASYL Database *Conference on Data Systems Languages.* The folks that brought you COBOL as well as the CODASYL standard for databases using the network model of data management. The network model consists of a series of records, with pointers to other series of records. It differs from the hierarchical model in that the network of pointers does not have to be strictly hierarchical.

code management system Software used to coordinate access to program files to ensure that programmers do not both try to simultaneously change a single file.

columns A table in the database has several columns, each one representing a particular piece of information in the table.

command line options When a program is executed from the operating system prompt there are typically several optional parameters. For example, QBF can be called in append mode using the -mappend command line option.

Common SQL A version of SQL used by Relational Technology for gateways that consists of a common subset of the versions of SQL used by the most prominent vendors of database systems.

communications server A computer whose primary purpose is to provide communications services.

complex data type A column in a Postgres database that has several pieces of information. An example of a complex data type is a column of type procedure which can return several pieces of information.

complex objects An object on a form that itself has several objects. A form is a complex object with fields and table fields.

complex queries A query involving several database tables.

concatenated key A key for a table composed of several columns. The key for an employee table might thus be the combination of first and last name.

concurrency Several users all accessing the same object, such as a database table.

consistency point Term used by the archiver and recovery processes in INGRES. When all transactions up to a certain point have successfully completed, the recovery process writes a consistency point into the log file, indicating to the archiver that it can remove those transactions to the journal files.

context level Term used in INGRES/team*work* data flow diagrams. The context level is the highest level of the model and shows all terminators that are objects outside the scope of the present analysis.

control flow A term used in INGRES/team*work* for modeling real-time processes. A control flow might be used to model the action of monitoring an indicator to see if it reaches a certain level.

control specification Related to control flow. A control specification, like a process specification, shows all of the steps to be taken when a particular control action is taken.

controlled redundancy Adding redundant data to the database for the purpose of decreasing the amount of time and the complexity for retrieving information.

coordinator database An INGRES database used by INGRES/STAR to store information about links to other components of the distributed database. The coordinator database can store any data that a local INGRES database can.

copyform An INGRES utility to move a form from one database to another.

CPU *Central processing unit.* You know—the computer part of the computer.

Cray A supercomputer.

create table The INGRES command used to create a new database table.

crosstab reports A report where the rows of data in the database are shown as columns on the report.

cursor An indicator on the display which shows the current position of a particular input device, such as a mouse or keyboard.

database A structured collection of tables available to the user through a query language such as SQL or QUEL.

database administrator The creator of a database. The administrator is responsible for defining security and other physical aspects of the database. The administrator is also responsible for performing backups and audits.

DataBrowser A Simplify utility for viewing data on a workstation.

data definition language The portion of a query language used to define new tables, define security constraints or modify the physical characteristics of the tables.

data dictionary A set of tables in a database or file system that hold data about data. For example, INGRES has a data dictionary containing the definition of all tables in a database.

data flow diagrams A graphical representation of an information system showing the processes, data stores and the flow of data between them.

Data Manipulation Facility The lowest level of an INGRES data server. The Data Manipulation Facility interacts with the file system on a computer to retrieve data from disk.

data repository Any file or database where information is kept. The repository is the actual data, as opposed to the data server that is the program used to access the repository.

data server The portion of INGRES that responds to SQL or QUEL requests and returns data.

data set Each table field in INGRES has a data set associated with it. The data set holds all rows of data, including those not visible to the user. When the user scrolls down the table field, information in the data set is displayed on the screen.

date A data type in INGRES. A column in a table can be defined as being of type date.

date arithmetic INGRES is able to perform mathematical operations on two pieces of data of type date. For example, the user can request all information in the database where a column date hired has a value less than “today” minus “1 year.”

DB2 An IBM database package based on the relational model and the SQL query language.

DBA See database administrator.

DBMS *Database management system.* Software that allows the centralized storage of data with multiple concurrent users, access control, and the use of a high-level data manipulation language such as SQL.

Deadlock Also known as a "deadly embrace." An example of deadlock is when two users each have a lock on a piece of data and are each waiting for a lock on the other user's piece of data.

DEC *Digital Equipment Corporation.* Makers of VAX computers, the VMS operating system, and the Rdb database software.

DECnet An implementation of the Digital Network Architecture by DEC, as opposed to implementations of DNA by other vendors.

decomposition The process of breaking a table up in the database into multiple tables. Horizontal decomposition, for example, might consist of moving all historical records in a sales table into a table called sales_history.

default A value or action that occurs when the user has not specified a choice. A default form is built in a QBF retrieval if the user has not supplied one built in VIFRED.

detail A term in the Report Writer that signifies a series of actions to be taken for every row of data retrieved from the database.

DIF *Data Interchange Format.* A format for files used on PCs that allows data from one application to be imported into another.

direct I/O A term used in VMS to refer to disk I/O operations. Contrast to buffered I/O.

distributed data environment A computing environment with data residing on different computers and different types of data repositories.

distributed database A single logical view of several data repositories. The distributed database looks to the user like a single database but is in fact a collection of several different data repositories.

distributed network A computer network with many different computers, each performing a specific task.

DMF See Data Manipulation Facility.

DMFCSP *Data Manipulation Facility cluster service process.* An INGRES process used to coordinate the integrity of logs across different nodes of a VAX Cluster.

DNA *Digital Network Architecture.* A network architecture developed by DEC that allows large networks of computers to be connected together.

DSRI *Digital Standard Relational Interface.* A DEC architecture for communication between front- and back-end systems.

duplicate keys Two rows in a table that have the same value for the key columns.

dynamic SQL SQL that is generated by a program at run-time as opposed to being hard-coded into the application.

EBCDIC *Extended Binary Coded Decimal Interchange Code.* A character code scheme used in IBM environments. See ASCII.

EMA See Enterprise Management Architecture.

embedded SQL INGRES allows SQL statements to be embedded into a 3GL. This allows the programmer to use the services of the database for I/O instead of defining and manipulating files.

end users Somebody who uses an application as a means for making decisions, as opposed to a programmer or application developer who develops tools for the end user.

Enterprise Management Architecture A DEC architecture for network management user interfaces that can work with multiple displays and protocols.

entity-relationship An information modeling tool that breaks an information system up into a series of entities that have relationships to each other.

environmental permit A form of security mechanism in INGRES that grants user's access to data based on the name of the user, the day of the week, or other environmental information. Contrast with data-valued permits, which grant access to data based on the value of the data in the database.

environmental variables A concept used on operating systems to provide customization of the user environment. A program consults an environmental variable to find the location or value for a particular function. For example, INGRES uses environmental variables as a mechanism to let each user see the representation of date that is most appropriate for that particular country. INGRES front-end processes look at the value of the variable to determine the proper way to display a date.

ESQL See embedded SQL.

Ethernet A data link protocol jointly developed by Intel, Xerox, and DEC and subsequently adopted by the IEEE as a standard. Several upper layer protocols, including DECnet, TCP/IP, and XNS, use the Ethernet as an underlying data link. Ethernet is to be contrasted with other data link protocols such as the token ring, DDCMP, or SDLC.

exclusive lock A lock on data that prevents other users from accessing it. Used for write operations. Contrast to a shared (read) lock.

executable image A program that is ready to run on an operating system. A program starts as source code and gets compiled to generate object code. The object code is then linked to form an executable image.

extensible data manager A data manager that allows users to add new data types, operators, aggregates, functions, or access methods.

FADS *Forms application development system.* A research project at the University of California at Berkeley under the direction of Professor Lawrence A. Rowe that led to the development of the INGRES forms-based interface.

fast commit A performance enhancement in INGRES that allows a transaction to be committed as soon as the transaction log is flushed to disk, instead of waiting for the data pages to be flushed to disk.

fault tolerance Fault tolerance is an attribute of a computer system that reflects its degree of tolerance to hardware and software failures while continuing to fun.

FDDI *Fiber distributed data interface.* A 100-mbps fiber optic local area network standard based on the token ring.

field An area on a display for user input and the display of data.

field activation A block of code in INGRES 4GL that is activated whenever a user tabs off of a particular field on the form.

file system The portion of a computer's operating system that is responsible for storing and retrieving pages of data onto a disk.

fill factor When a table is modified to a new storage structure, the fill factor parameter reflects the amount of space to be left on data pages for the addition of data at some future time.

first normal form A database table is in first normal form if there are not two or more columns for one piece of information. Two or more columns for the same information is known as a repeating group.

floating point A native data type on most operating systems. A floating point number is one that can have numbers after the decimal point, in contrast to an integer that cannot.

footer A term used in the Report Writer. A footer action occurs at the termination of a break action. For example, at the end of each page, the page footer section of the report could specify that the page number and current date be printed.

Forms Run-Time System The INGRES component responsible for managing forms on all forms-based user interfaces.

fourth-generation language A group of languages often linked with database packages such as INGRES. Contrast with FORTRAN and other third-generation languages.

frame An object in the INGRES user interface that consists of a form and a menu. Frames can be defined by the user in ABF and are also present in other front ends such as QBF.

front end A front end is a program that a user interacts with. The front end then sends off requests to the back end for data.

FRS See Forms Run-Time System.

FRSkey *Forms Run-Time System Key.* A logical key definition, such as HELP. A programmer in ABF can designate a block of code that is executed whenever the HELP logical key is activated. The FRSkey is then mapped to a specific physical key on the terminal at run time.

full sort merge A strategy for joining two tables used by the query optimizer and query executor in INGRES. A full sort merge sorts both tables involved in the join and then starts comparing records in each table looking for a match.

function A function takes as input a piece of data and returns a value. For example, the query language has functions that can accept a date and return the day of the week that the date falls on.

Gateway An INGRES program that serves as a bridge to another vendor's file system or DBMS. Gateways allow INGRES users to treat a foreign data repository as an INGRES database.

GCA See GCF application interface.

GCF See General Communication Facility.

GCF application interface *General Communication Facility application interface.* The top layer of the General Communication Facility. GCA is a library of routines built into every INGRES component. GCA allows INGRES components to be unaware of the communications protocols used to reach their peer component.

General Communication Facility The architecture used by INGRES to mask the details of communicating across a heterogeneous network. GCF allows any front end to communicate with any back end on the network.

general-purpose user interface An application not tied to a specific database or type of user. QBF is a general-purpose user interface because it can be used to append, retrieve, and update data on any table in an INGRES database.

gigabytes *billion bytes of data.*

granularity A term used in the lock manager. Granularity refers to the amount of information that a lock affects. A database lock has a very coarse granularity, while a page-level lock is of fine granularity.

group commit An INGRES performance enhancement that allows a single I/O operation to be used to commit several transactions to disk. Batching transactions up into a single transaction means that fewer I/O operations are needed.

hash A storage structure in INGRES. A hashed table takes the key value for a record and performs a mathematical transformation on it to locate the appropriate page of data. Contrast with BTREE and ISAM storage structures that use an index to locate the appropriate page.

header A Report Writer concept. The header section for a break column is executed whenever the value of the break column changes. A header action for a page break, for example, might be used to print column headings.

heap A storage structure for data where data are not placed in any particular order, requiring a scan of the entire table for every retrieval.

heap sort A heap sort sorts the data before placing it in a heap. If a user is requesting data in sorted order, there is a good chance that it will come off the heap in the proper order. As more data are added to the bottom of the heap, the data degenerates into non-sorted order.

help_frs A command in the INGRES 4GL that activates the help subsystem.

heterogeneous Different.

heterogeneous network A network consisting of different network protocols or kinds of computers. A network combining SNA and DNA protocols using an SNA gateway to connect the two is a heterogeneous network.

hidden fields A field in the INGRES 4GL not visible to the user. Equivalent to a program variable in a third-generation language.

hierarchical database A database that structures data as a hierarchy instead of in tables. Programmers then navigate the hierarchy to retrieve a particular row of data. IMS is an example of a hierarchical database system.

Hierarchical Storage Controller Stand-alone disk and tape controller used in clusters using the CI bus. The HSC is actually a modified PDP computer that has been optimized as a mass storage controller.

histogram A histogram groups data into ranges and shows how many pieces of data fall within each range. Histograms are used by the query optimizer to determine what percentage of a table might meet a particular qualification on a query.

homogeneous The same.

HSC See Hierarchical Storage Controller.

icon A small pictorial object on a workstation used to represent a closed window. The user points to the icon, clicks the mouse button, and a window opens.

IIDBDB *INGRES Database Database.* The master database on an INGRES installation that keeps track of users, databases, and locations.

IIMONITOR An INGRES utility used to monitor the status of different data servers.

IMS *Information Management System.* Database management software from IBM based on the hierarchical data management model.

inconsistent database A database with missing or corrupted tables. The job of the recovery manager in INGRES is to prevent an inconsistent database.

index A direct access method to data. An index has a key value and a pointer to the row of the table that contains data with the key value. An index can be a primary index, where the index is part of the storage structure of the actual table, or a secondary index that is a separate table in the database with pointers to the base table.

indexed sequential access method A file structure that allows random access to data via an index and then sequential access to data after that.

Inference Corporation Makers of the ART artificial intelligence software.

information architecture A collection of tools that allow the integration and management of data in a complex, heterogeneous network.

information model INGRES/team*work* term for an entity-relationship diagram.

INGMENU See INGRES/MENU.

INGRES A popular relational database management system that runs on a variety of operating system platforms.

INGRES/Gateway An INGRES program that is able to access non-INGRES data repositories such as a DB2 relational database or a VSAM file.

INGRES/MENU An INGRES program that serves as a shell for access to the other subsystems. The user can select a variety of INGRES functions from INGRES/MENU and the subsystem is called.

INGRES/NET The INGRES program that allows front and back ends to communicate across a heterogenous network.

INGRES/STAR The INGRES program that allows several different databases to appear as a single, logical, distributed database.

inheritance A Postgres concept that allows a table to inherit columns and rows from another table. A row in the employee table, for example, could inherit more general information from a people table such as telephone number or sex.

inittable An INGRES 4GL command used to initialize a table field and its associated data set.

instruction set The set of low-level commands on a computer. When a user issues a command, it is ultimately translated to a series of commands within the instruction set. Assembly language program allows direct access to these low-level commands.

integer A basic data type on a computer consisting of numbers without any decimal places.

integrity A concept in database systems that refers to keeping data consistent. An integrity definition for a state abbreviation in an address table might consist of saying any data in the table must be in a list of fifty valid abbreviations.

IntelliCorp Maker of the KEE artificial intelligence software.

interface A well-defined set of commands that are used to interact with a program. SQL, for example, is the interface between a front- and back-end process in a relational database system.

interprocess communication The facility on a computer that allows one program (process) to communicate with another one on the same computer system.

I/O *Input/output.* The process of moving data from disk to main memory and back again.

IPC See interprocess communication.

IRD *Information Resources Dictionary.* The data contained in a data dictionary that conform to the IRDS standard.

IRDS *Information Resources Dictionary System.* An ANSI and ISO standard for data dictionaries and the operations used to access the data dictionary.

ISAM See indexed sequential access method.

ISDN *Integrated Services Digital Network.* A new international communications standard that allows the integration of voice and data on a common transport mechanism.

ISO *International Organization for Standardization.* International standard making body responsible for the OSI network standards and the OSI Reference Model.

iterative query A type of query in Postgres that allows the query to keep on running until no more data are affected by the command.

join A join combines two tables in a database based on some common value that the two tables share. A table with employee definitions might be joined with another table on project definitions. The join column would be the employee name.

JoinDef See join definition.

join definition A QBF concept used to define how two or more tables are joined together. The join definition includes which columns are used for the join as well as update and delete rules.

join nodes An operation used in the INGRES query optimizer. A join node signifies that two tables are being joined together (as opposed to a sort node or project-restrict node).

journal A file that contains a list of the operations that occurred on a particular database. Journals are used for recovery purposes and audits on the database.

KEE Knowledge engineering software.

KEEconnection Software made by Intellicorp used to map data between KEE knowledge databases and INGRES databases.

key activation A type of command in the INGRES 4GL. A key activation says that when a user hits a particular key on the keyboard, a block of INGRES 4GL code should be activated.

knowledge base The database associated with an artificial intelligence environment such as KEE or ART.

LAT See Local Area Transport.

leaf level A part of a file using the BTREE storage structure. The leaf level has one entry for each of the records in the table. The leaf level then has a pointer to the page actually containing the data.

library A set of functions accessible from a program. The library is designed to be used by several different programs. Mathematical functions such as square root might be put into a math library and then linked in with the program.

life-cycle control facility A part of the IRDS data dictionary standard used to govern when data is moved from one life cycle partition to another.

life-cycle partition All objects in the IRDS data dictionary are kept in a life cycle partition. A partition shows the stage of development, such as uncontrolled or public, of that object.

linking The process of taking several different subprograms and libraries and combining them into a single executable image or program.

livelock Occurs when a user requesting an exclusive lock on an object is waiting behind a user who already holds a shared lock. Subsequent users who request a shared lock are granted their request, meaning that the requester of the exclusive lock could wait indefinitely.

Local Area Transport A DEC architecture for terminal servers on Ethernet networks designed to conserve bandwidth and offload processing from hosts.

location An INGRES concept that allows a portion of the INGRES environment, such as a particular database, to be stored in different disk drives on the computer. Each of these disk drives is one, or several, locations.

location transparency A concept in distributed databases that says that the user of the database should be unaware of the location of the data.

locking The process of indicating that a particular object, such as a page in a database table, is in use to prevent other users from performing incompatible actions on the object. A lock can be shared, as in the case of read access, or exclusive, as in the case of write access.

log file An INGRES file containing all active and recently completed transactions. The log file is used by the recovery manager in case of aborted transactions.

logical design The logical design of a database consists of tables as seen by users. The logical design is then translated to a physical design by adding views, indices, storage structures, security, and other implementation details.

logical name A VMS environmental variable.

look and feel A common method of interacting with a computer across different programs. The INGRES forms system is an example of a look and feel standard for terminals; the Open Look standard is a look and feel standard for bit-mapped workstations from vendors such as Sun Microsystems.

Lotus 1-2-3 A popular PC-based spreadsheet.

LU0 *Logical Unit Type Zero.* A class of programs in the IBM System Network Architecture network that essentially require each of the programs to perform low-level functions. Used for applications needing high performance and control over operation on the network.

LU6.2 *Logical Unit Type 6.2.* A class of functions in IBM's SNA that provides program to program communications. Sometimes known as Advanced Program to Program Communication (APPC).

Macintosh A computer made by Apple Computer. The Macintosh is characterized by the graphical, intuitive user interface.

Macro Assembly language for a VAX.

magnetic disk The most common form of secondary storage for a computer system.

main memory Also known as random access memory (RAM) or core memory, this is the primary storage mechanism for a computer. Programs, the operating system, and data all reside in main memory when being accessed by the CPU.

mapping file An INGRES file that maps logical Forms Run-Time System functions to keys on a particular keyboard.

master-detail query When two tables are joined together, each row in one table (the master table) will have several rows associated with it in the other (detail) table. Used in QBF and the INGRES 4GL.

Mbyte See megabyte.

mbps *million bits per second.*

megabyte *million bytes of data.*

menu A series of options available to the user.

message An INGRES 4GL command used to put a text string on the screen to communicate with the user.

meta-data Data about data. The IRDS data dictionary or the INGRES system catalogs both contain definitions of data. A program would consult the meta-data and then go find the data that is described.

meta-schema The schema for meta-data. The schema describes how the meta-data is stored.

method A term used in object-oriented programming. Each object has a series of methods associated with it. A form in INGRES, for example, would have the method "display form" associated with it.

MicroVAX A series of DEC processors using the Q-bus and competing in the workstation market with Sun and Apollo.

MIP *Million instructions per second.* A measure of the speed of a CPU.

MIS *Management information system.* An application used to provide information to managers in an organization. The term has come to refer to the department in an organization responsible for computing.

MIT *Massachusetts Institute of Technology.* Developement hub for the X Windows System. Also a university.

model An abstract representation of a real-world process. A user might define an entity-relationship diagram as a model for a database.

modify An INGRES command used to change the storage structure of a table.

MONITOR A VMS tool used to examine the current status of a system.

mouse A pointing device used on workstations.

multistatement transaction Several different interactions with the database that are grouped into a single transaction. If any one of the operations is not carried do to a user abort or system crash, the entire transaction is rolled back. A multistatement transaction has the characteristic that either all or none of the operations will be carried out.

Multiplex A software product made by Network Innovations (owned by Apple) that retrieves information from a variety of VAX database packages and translates it into a variety of different PC formats.

multi-server architecture The INGRES architecture for data managers that allows several different servers, as in the case of a parallel processor or VAX cluster, to all access the same database.

multivolume table A database table that is split over several disk drives. This might be because of the size of the table or because using several disk drives is potentially faster then only using one.

MVS/TSO *Multiple virtual storage/time sharing option.* MVS is an IBM operating system. TSO is the interactive subsystem, as opposed to a system like JES used for batch processing.

name server A program that translates a name into an address. The INGRES name server allows a front-end process to find the location for a particular back end.

namespace The collection of names in a certain environment. A data dictionary might be one namespace.

natural language An interface that allows a user to use English instead of a structured language such as SQL.

Natural Language Interface A product made by Natural Language, Incorporated, that allows a user to issue English-language requests for data in an INGRES database.

nested dot notation A Postgres concept that allows users to access subobjects of a complex object. For example, a table has a column, which in turn contains several columns.

network A series of computers connected together.

network architecture A carefully defined set of functions and the interfaces between the functions that allow any two programs that implement the architecture to communicate.

Network Innovations A subsidiary of Apple Computer and makers of the Multiplex and CL/1 products.

network layer The third layer of the OSI Reference Model. The network layer is responsible for delivering a packet of data to the destination within the network.

NLI See Natural Language Interface.

NLI Connector The product made by Natural Language, Incorporated, used to define the schema and semantical content of data in a database to the natural language parser.

Non-SQL gateways An INGRES program that allows the use of SQL on nonrelational query targets, such as a hierarchical database or a file system.

normalization A database design technique used to reduce update anomalies by breaking up a logical design into several tables.

null A special value for most data types that indicates no data is present.

numeric template A formatting technique used in forms systems and report writers to show how to display the data.

on-line recovery A technique for recovering from aborted transactions and system failures automatically, without requiring the database administrator to shut down the database.

Open Look A look and feel standard for bit-mapped workstations developed by Xerox and adopted by AT&T and Sun Microsystems.

Open Systems Interconnect The International Standards Organization's network architecture based on the OSI Reference Model.

operator A part of a query language. Operators are used in the qualification of a query to define how a comparison should function. Plus, equal, and greater than are all examples of operators.

optical disk A form of tertiary storage that allows large amounts of data to be stored on a disk.

optical disk jukebox A device that allows a computer to access many different optical disks.

optimizedb An INGRES program that constructs a profile of data in the database, which is then used by the query optimizer to decide among different access strategies.

OSI See Open Systems Interconnect.

OSI Reference Model A seven-layer protocol stack with a standard set of functions and interfaces used as the model for OSI and other network architectures.

OSL *Operations Specification Language.* Another name for the INGRES 4GL.

overflow page A page of data not directly referenced by an index in ISAM or the hashing algorithm for hashed tables. The index points to a primary page, which in turn has pointers to overflow pages.

packet A general term used in networking to refer to a message sent to a peer entity in the network.

packet switching A type of data communications network that allows many users to share a single (or several) physical lines. Opposed to circuit switching that allows a user to set up a dedicated circuit for use in communications.

page The fundamental unit of I/O on a file system or database system. A page of INGRES data is 2048 bytes and can contain one or several rows of data.

panel interface An interface to the IRDS data dictionary that is menu-driven.

parallel processor A computer with several CPU's that share peripheral devices such as disk drives.

parse The process of converting a stream of input into a series of tokens, or parts of the language.

partial key search A query on a table that uses only a portion of the key. For example, if last name is the key for a table, a search on all names beginning with a capital M is a partial key search.

PC *Personal computer.* IBM series of computers or clones.

peripheral device A device connected to a computer system, such as a disk drive or a terminal.

permits A series of rules in an INGRES database that define which users may access which tables and under what conditions.

PF2 *Programmable function key 2.* General-purpose function key on a terminal. PF2 is used on VT100 terminal, equivalent to F2 on a PC keyboard.

physical design The process of deciding how to store data in a database. The physical design stage, in contrast with logical design, concentrates on physical storage structures, permits, and other aspects of the system that are typically transparent to the user.

Picasso A research project at the University of California at Berkeley, under the direction of Professor Lawrence A. Rowe, that is investigating extensions to user interfaces and programming techniques for a workstation-based environment. Also an artist.

pop-up An INGRES form, message, or prompt that is displayed at a particular location on the screen without destroying existing data.

portability The ability to move source code, such as INGRES 4GL, to another computer system without modifying the code.

Postgres A database research project at the University of California at Berkeley investigating extensions to the relational model.

POSTQUEL The query language used in Postgres.

PostScript A page description language developed by Adobe Systems used in many laser printers and as a display mechanism for some computers from Sun, DEC, and NEXT.

precompiled query A query, having already been submitted, that is already compiled. A precompiled query executes quicker because several steps, such as parsing the query, are bypassed.

precomputed query A performance mechanism in Postgres that saves the result of a previous query. If the query is submitted again, the query does not have to be run since the answer is already known.

PreJoin Information in Simplify that specifies how tables are to be joined together, saving the user from having to specify this information.

presentation layer The sixth layer of the OSI Reference Model that translates information into a format that is recognizable by both machines that are communicating.

Presentation Manager The portion of IBM's System Application Architecture responsible for the user interface.

primary key The columns of a table that are used by the storage structure to store the data.

primary pages Those pages in a table that are directly pointed to by an index or hashing algorithm. Overflow pages are created when the primary pages are full. The data manager first goes to the primary pages, then pulls up all overflow pages.

procedure A collection of query language statements maintained in the back end.

process A standalone program that runs on a computer. A complex program may be made up of several processes.

process specifications The variables and processing steps in the process portion of a data flow diagram.

projected A term used in relational database theory to denote the process of selecting only certain columns from a table.

project-restrict nodes A phase in a query execution plan that takes out unneeded columns (projects) and unneeded rows (restricts).

protocol stack A series of processes in a network architecture, each providing a service for the process directly on top of it. This layering mechanism allows functionality in a lower layer of the protocol stack to be rewritten without rewriting the other layers.

prototyping Quickly developing a version of a program to determine the feasibility and the user reaction. Prototypes are then refined into production applications.

public domain Intellectual property available to people without paying a fee. Most computer software developed at universities is in the public domain.

QBF See Query-By-Forms.

QBFname A pairing between a query target (a table, view, or JoinDef) and a form.

QEP See query execution plan.

QRYMOD *Query modification.* A process in the INGRES back end that modifies a query to add integrities, permits, and translates views into their definition.

qualification The portion of a query statement that qualifies which rows of data a user is interested in. Restricting a retrieval to all rows where salary is less than $20,000 is an example of a qualification.

quality indicators A term used in the IRDS data dictionary that allows user to label an entity with an indicator of the quality of the definition. Quality indicators are used to supplement life cycle phases with a more granular indicator.

QUEL *Query language.* The original query language used in INGRES.

query A request for data.

Query-By-Forms INGRES program used to browse and change data in a forms-based application.

query execution plan The series of steps that will be taken to find the results of a query.

query optimizer The INGRES program that decides on the best query execution plan out of all the different possibilities.

query target The object in a database, such as a table, JoinDef, or view, that an application will run against.

Rally A DEC user interface for Rdb databases.

RAM *Random access memory.* Dynamic memory, sometimes known as main memory or core.

RAM disk A portion of main memory that is allocated as a disk drive instead of more traditional functions such as working sets for users.

RDA See Remote Data Access.

Rdb DEC's relational database management system.

read ahead The process of reading extra, unrequested pages of data and caching them in anticipation of future requests for those pages.

records Files are divided into a series of records, normally corresponding to one line of text or data.

recovery manager INGRES program that keeps the database consistent when a system crashes or transaction aborts.

reduced instruction set computer Generic name for CPUs that use a simpler instruction set than more traditional computer architectures. Examples are the IBM PC/RT, Pyramid minicomputers, and the Sun 4 (SPARC) workstations.

referential integrity A set of rules that specifies the relationship between one database table and another.

relation Another word for a database table.

relationship A component of an entity-relationship diagram. The relationship shows how two entities are connected.

Remote Data Access ISO protocol for remote access to SQL-based relational databases.

repeating values Database design construct where several occurrences of a variable are stored as separate columns in a table, rather than as additional rows. Repeating values are considered to be a violation of most relational database design methodologies.

report INGRES command that runs a report stored in the system catalogs.

Report-By-Forms Forms-based method of defining INGRES reports.

Report Writer Command language-based method of defining reports.

restriction Term used in relational database theory for removing certain rows from a table.

resume INGRES 4GL command that exits the current block of code and puts the user back into a display loop.

retrieve QUEL command to get data from the database.

reverse video Highlighting a field on a screen so that it stands out.

RISC See reduced instruction set computer.

RMS *Record management services.* A common I/O interface for VMS used for access to local data via QIO calls and remote data via the DAP protocol.

rollback To reverse the effects of an operation on the database, as in the case of reversing the first few steps of an aborted multistatement transactions.

rollforwarddb An INGRES command that applies changes stored in a journal file to a database in the case of having to recover the database from a backup.

RS/1 A data modeling environment made by BBN.

RTI *Relational Technology, Incorporated.* Makers of INGRES.

rule A Postgres concept that allows the user to formulate rules that become part of the database. Whenever a condition specified by the rule is met, an action or series of actions will occur.

run-time data selection Deciding, when a report is run, exactly what data should be retrieved, instead of building the entire data specification into the report when it is designed.

SAA See System Application Architecture.

SAS *Statistical Analysis System.* A program made by the SAS Institute; used frequently for complex statistical analysis.

SCA See System Communication Architecture.

scan mask An IRDS concept used to select which objects are to be retrieved.

scanning The process of reading every row in a database table instead of using an index to selectively read rows.

schema The definition of the tables that make up a database.

scroll To move up or down, as in to scroll up the rows of a table field.

second normal form A database design principle that requires that every nonkey column in a table is directly or indirectly dependent on the key.

secondary index A table in the database that has a key value and a pointer to the rows in a primary table that have that key value. The secondary index is used to supplement the primary storage structure of a table.

select An SQL statement used to retrieve data from the database.

selection criteria The part of a SQL select or QUEL retrieve statement that indicates which rows should be retrieved.

Sequent A brand of parallel processor.

sequential key A condition that occurs when all the values for a key column increase sequentially, as in the case of some purchase order numbers.

server Any program or computer that provides a service to other programs or users. A data server, for example, provides data to front-end programs. A terminal server provides dedicated hardware and software for the purpose of giving terminals access to the network.

service advertisement A part of DEC's Local Area Transport architecture. All nodes that are able to provide a particular service periodically advertise the availability of that service and a service rating. The terminal server then logs the user onto the node with the best current service rating.

services interface An access mechanism to an IRDS data dictionary. The services interface consists of a series of library calls that can be used to add entities or other IRDS operations.

session layer The fifth layer of the OSI Reference Model. The session layer maintains a session between two users.

shadowed A disk drive is shadowed when two copies of the data are kept on two separate disk drives. Whenever a piece of data is changed, it is changed on both copies. If one disk drive fails, the shadow is available as an instantaneous backup. Sometimes known as mirrored disk drives.

shared lock A lock that can be shared among multiple users. Two users reading data can share a read lock. A write lock, however, cannot be shared so it is an exclusive lock.

shared object hierarchy A hierarchy allows one object to inherit characteristics from a higher level. A shared object hierarchy allows the objects to be shared among multiple users.

simple field A field in the INGRES forms system that can only have one value, as opposed to a table field, which can have several.

Simplify A workstation-based interface to INGRES developed by Sun Microsystems and enhanced by Sun and Relational Technology.

singleton query A query in the INGRES 4GL that only retrieves one row of data.

SNA See System Network Architecture.

sort nodes Nodes in a query execution plan that sort the data. Often occurs before a join node or at the completion of the query.

source files Files containing source code, such as INGRES 4GL code. The source files are then compiled into object files that contain machine language code. Finally, the object files are all linked together to form an executable image.

spawning Creating a new process to run on a computer system.

spreadsheet A program that allows users to establish relationships between rows and columns of data in a tabular format.

SQL See Structured Query Language.

sreport INGRES command used to load a file containing report specifications into the system catalogs.

standard catalog interface A portion of the INGRES system catalogs used by the front-end processes. The standard interface allows the front end to be unaware of the actual location or format of the data and thus hides the presence of distributed databases and gateways.

storage structure The way data in a table is actually stored, such as ISAM, BTREE, or hashed.

store A term used in data flow diagrams to refer to files, database tables, or other data repositories.

structure chart A graphical representation of the design of a information system.

structured design A methodology for the design of information system that breaks the program down into a series of modules with carefully specified interfaces between the modules.

Structured Query Language ANSI standard data manipulation language used in most relational database systems.

submenu An INGRES 4GL that allows a menu to appear inside of another menu. Often used to control the retrieval of multiple rows of data from the database.

subsystems Any portion of a larger program. QBF is a subsystem of INGRES.

Sun Microsystems Develop of Sun Workstations, the Network File System, and the Simplify user interface for database systems.

syntax The specification of a language.

sysgen A DEC or IBM program used to set operating system parameters.

sysmod An INGRES command used to remodify the system catalogs to increase performance.

System Application Architecture An IBM architecture used to bring together the diverse operating systems and hardware platforms used in IBM environments. SAA consists of a common user interface, programming interface, and communication interface.

system catalogs A set of tables in an INGRES databases used to manage the database. The system catalogs are a data dictionary.

Systems Communications Architecture DEC's network architecture for VAX Clusters.

system crash When a computer stops running, as in the case of a power failure.

system files Files used by the operating system or a program, as opposed to user files that contain user data.

system manager The person responsible for maintaining and administering a computer system.

System Network Architecture IBM's network architecture.

table An object in a relational database system composed of rows and columns.

table fields An object in the INGRES forms system that has several columns and rows of data displayed at the same time.

TCP *Transmission Control Protocol.* A transport layer protocol in TCP/IP that provides reliable end-to-end communications.

TCP/IP *Transmission Control Protocol/Internet Protocol.* Department of Defense-sponsored family of networking protocols, used frequently in Unix environments.

Teamdata DEC-developed user interface for DSRI compatible relational databases.

team*work* A collection of computer-aided software engineering (CASE) tools developed by CADRE and enhanced in a joint development project between Relational Technology and CADRE.

team*work*/IM *teamwork/Information Modeling.* A CASE tool for developing entity-relationship diagrams.

team*work*/RT *teamwork/Real-time.* An enhancement to team*work*/SA for real-time analysis.

team*work*/SA *teamwork/Systems Analysis.* A CASE tool for developing data flow diagrams.

team*work*/SD *teamwork/System Design.* A CASE tool for structured design of information systems.

temporary tables A table used for the temporary storage of data. Temporary tables go away at the end of the current session.

termcap file *terminal capability file.* A file that contains the capabilities of different terminals and the physical commands used to activate those capabilities.

Terminal Monitor An INGRES program that allows the user to directly enter QUEL or SQL statements.

terminators Term used in data modeling to refer to entities outside the scope of the current analysis.

tertiary storage Third-level storage such as optical disk or magnetic tape. Contrast with primary storage (main memory) or secondary storage (magnetic disk).

ThinWire Thinner, and cheaper, version of baseband coax cable used for Ethernet networks. Also called CheaperNet.

third-generation language Traditional programming languages such as BASIC, FORTRAN, or COBOL.

TID See tuple ID.

timeout A parameter that indicates how long an operation should be queued before returning an error message. INGRES, for example, offers a timeout parameter for users waiting for a lock.

token ring A data link protocol frequently used in PC-based networks. Ethernet is another data link protocol.

TP4 *Transport Protocol Class 4.* One of five classes of transport layer protocols that is in the Open Systems Interconnect network architecture. TP4 is the most functional of the five classes, providing reliable end-to-end communications. TP4 is modeled after the TCP protocol in TCP/IP.

transaction A database operation or set of database operations that together form a single transaction. The database ensures that either all or none of the statements will take effect.

transitive closure A type of Postgres query that allows the query to continue running until no more data are retrieved or affected.

transport layer The fourth layer of the OSI Reference Model. The transport layer is responsible for providing reliable end-to-end communications.

transport service The entity in a network implementation that provides the fourth layer of the OSI Reference Model. In DECnet, this is the Network Services Protocol (NSP); in TCP/IP it would be the Transmission Control Protocol (TCP).

trigger A Postgres concept that allows a series of statements to be executed (triggered) whenever a certain condition occurs in the database.

trim Text on a form not associated with a field.

tuple A term used in relational database systems. A tuple is the equivalent of a record in a file management system and corresponds to one row of data in a table.

tuple ID An identifier for each tuple in a database table. The tuple ID consists of the page that the tuple resides on together with the offset of the tuple on the page.

twisted pair A pair of wires (or several pairs of wires) such as is used to connect telephones to distribution panels. Twisted pair is also being used as a physical transmission media for Ethernet, token ring, and other forms of data links.

two-phase commit protocol A protocol used in distributed databases that ensures that all portions of a multistatement transaction are successfully completed or none are completed.

University INGRES The original version of INGRES developed at the University of California at Berkeley.

Unix Operating system developed and trademarked by American Telephone and Telegraph. "Unix" is a pun on the Multics operating system.

unloadtable An INGRES 4GL command used to cycle through all the visible and nonvisible rows in a table field.

user interface What the user sees. QBF is an example of a user interface. The user interface then communicates with the back end for access to data.

vacuum demon The Postgres process that moves data from secondary to tertiary storage.

validation criteria A VIFRED concept that defines what constitutes valid data for a particular field on a form. If data entered violates the validation criteria, an error message is displayed to the user.

VAR *Value-added reseller.* Company that embeds another vendor's products into a more sophisticated product.

VAX *Virtual address extension.* A series of computers made by DEC.

VAX Cluster A high-speed network from DEC that allows several computers to share a single file system and other resources.

versioning A Postgres concept that allows several different versions of a single table to exist in the database.

vertical fragmentation Storing different rows of a database table in different places.

view A virtual table in the database. A view consists of an SQL select statement that defines a retrieval of data. The view then looks like a table to the user.

VIFRED See Visual-Forms-Editor.

VIGRAPH *Visual-Graphics-Editor.* The INGRES program used to create and edit graphs to display data.

virtual connection A connection on a network between two programs. The network allows several virtual connections can all share a single wire, or physical connection, between two computers.

visual characteristics The attributes of a form that are visible to the user. A visual characteristic would be highlighting a field on the form using reverse video.

Visual-Forms-Editor An INGRES program used to customize forms for use in QBF or other INGRES subsystems.

VM *Virtual machine.* An IBM operating system that permits guest operating systems, such as MVS, to reside on top of it. Usually used in conjunction with the CMS user interface.

VMS *Virtual memory system.* A DEC proprietary operating system for VAX computers.

VSAM *Virtual sequential access method.* File organization method used in IBM environments for direct access files. Similar to ISAM (indexed sequential access method).

VT-100 An intelligent terminal manufactured by DEC. VT-100 usually refers to the series of terminals, beginning with the VT-100 that uses the ReGIS protocols.

wide key A key for a file or database table that has many letters or numbers. A wide key increases the size of the index for a storage structure, resulting in slower access time to the data.

wild card A pattern matching symbol that matches any sequence of characters. The pattern M* would match any character string starting with the letter M.

window A portion of a display screen devoted to a particular task. One window might be used for an electronic mail program, a second to display data using QBF.

working set A VMS parameter that defines how big a part of main memory a particular process gets.

workstation A personal computer with a high-resolution bit-mapped graphics display screen usually connected to a network.

X Windows System A protocol developed at the Massachusetts Institute of Technology that defines how a program on a network is able to share the real estate on a display on a workstation with other programs.

X.21 CCITT standard for circuit-switched networks.

X.25 CCITT standard for packet-switched networks.

Index

(Continued from the front flap)

glean insights on database design and its connection to better information systems.

This book looks at the evolution of the relational DBMS and looks at key research projects that will affect the future of software systems like INGRES. Malamud shows how INGRES can be integrated with other software systems, with numerous computing platforms, and with complex, heterogenous networks.

INGRES: Tools for Building an Information Architecture is both a working handbook and a state-of-the-art look at the future of database technology. For data processing managers, it is the only one-stop resource that will help them maximize efficiency today and ensure information systems productivity through the 1990s.

About the Author

Carl Malamud has been involved with Relational Technology, Inc. and its INGRES relational database system from the system's inception in 1981. A certified INGRES instructor, he has frequently taught DP professionals about this system. He has also developed seminars on DEC computing and VAX-based database systems, which are frequently taught in the United States and Europe; and has authored *DEC Networks and Architectures.*

Mr. Malamud is a consultant specializing in networks and database management systems for government agencies, corporations, research laboratories, and universities. He is an MBA graduate of Indiana University.

VAN NOSTRAND REINHOLD

VAN NOSTRAND REINHOLD
115 Fifth Avenue, New York, NY 10003

ISBN 0-442-31800-6

90000

9 780442 318000